Helion

Microsoft

WYDANIE VII

Windows od środka

Wnętrze nowoczesnego systemu, wirtualizacja, systemy plików, rozruch, bezpieczeństwo i dużo więcej

Professional

Andrea Allievi
Alex Ionescu
Mark E. Russinovich
David A. Solomon

Tytuł oryginału: Windows Internals, Part 2, 7th Edition

Tłumaczenie: Andrzej Watrak (wstęp, rozdz. 8, 11 – 12), Piotr Rakowski (rozdz. 9 – 10)

ISBN: 978-83-283-9072-0

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/wiosw7
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 9.

Technologie wirtualizacji

Jedną z najważniejszych technologii służących do uruchamiania wielu systemów operacyjnych na tej samej maszynie fizycznej jest wirtualizacja. W chwili pisania tego tekstu dostępnych jest wiele typów technologii wirtualizacji, oferowanych przez różnych producentów sprzętu, które z biegiem lat ewoluowały. Technologie wirtualizacji są wykorzystywane nie tylko do uruchamiania wielu systemów operacyjnych na maszynie fizycznej, ale stały się również podstawą ważnych funkcji zapewniających bezpieczeństwo, takich jak **wirtualny tryb bezpieczny** (VSM — ang. *Virtual Secure Mode*) oraz **integralność kodu wymuszona przez hipernadzorcę** (hiperwizora) (HVCI — ang. *Hypervisor-Enforced Code Integrity*), których nie można uruchomić bez hipernadzorcy.

W rozdziale tym omówiliśmy platformę wirtualizacji zaimplementowaną w systemie operacyjnym Windows, znaną pod nazwą Hyper-V. Platforma Hyper-V składa się z hipernadzorcy (hiperwizora), który jest komponentem zarządzającym warstwą sprzętową wirtualizacji zależną od platformy, oraz stosu wirtualizacji. Opisujemy tutaj wewnętrzną architekturę platformy Hyper-V i przedstawiamy krótki opis jej komponentów (menedżera pamięci, procesorów wirtualnych, przechwytów, planisty [schedulera] itd.). Stos wirtualizacji jest zbudowany na warstwie hipernadzorcy i zapewnia różne usługi partycjom głównym (ang. *root partitions*) i partycjom gości (ang. *guest partitions*). Opisaliśmy również wszystkie komponenty stosu wirtualizacji: proces roboczy maszyny wirtualnej (VMWP — ang. *VM Worker Process*), usługę zarządzania maszyną wirtualną, sterownik wirtualizacji infrastruktury (VID — ang. *Virtualization Infrastructure Driver*), magistralę maszyny wirtualnej (VMBus — ang. *Virtual Machine Bus*) i pozostałe komponenty, oraz różne obsługiwane emulacje warstwy sprzętowej.

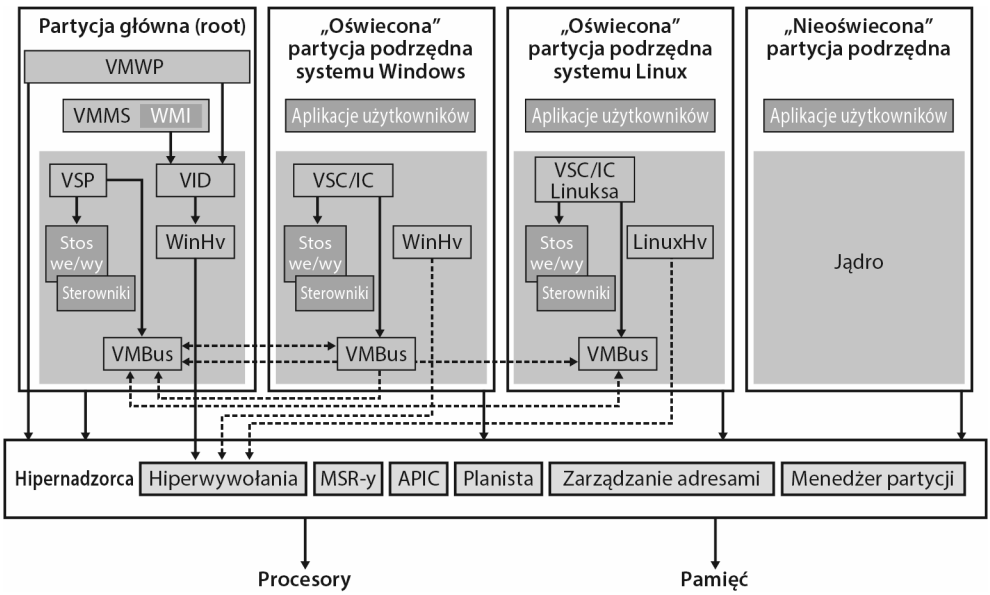
W ostatniej części rozdziału opisujemy niektóre technologie oparte na wirtualizacji, takie jak wspomniane wyżej VSM i HVCI. Przedstawiamy wszystkie usługi bezpieczne, które są dostarczane systemowi przez te technologie.

Hipernadzorca systemu Windows

Hipernadzorca platformy Hyper-V (znany także jako hipernadzorca systemu Windows) jest hipernadzorcą typu 1 (natywnym lub tzw. bare-metalowym), czyli minisystemem operacyjnym, który działa bezpośrednio na warstwie sprzętowej hosta — maszyny fizycznej. Jego funkcją jest zarządzanie pojedynczym głównym (ang. *root*) systemem operacyjnym maszyny fizycznej, umieszczonym na partycji głównej, oraz jednym lub wieloma systemami operacyjnymi maszyn wirtualnych gości umieszczonych na partycjach podrzędnych (ang. *child partitions*). W przeciwieństwie do hipernadzorców typu 2 (lub hostowanych), którzy działają na bazie

konwencjonalnego systemu operacyjnego jak zwykłe aplikacje, hipernadzorca systemu Windows tworzy warstwę abstrakcji separującą główny system operacyjny od maszyn wirtualnych. Główny system operacyjny „wie" o istnieniu hipernadzorcy i komunikuje się z nim, aby umożliwić uruchamianie jednej lub wielu maszyn wirtualnych gości. Ponieważ hipernadzorca jest częścią systemu operacyjnego, zarządzanie znajdującymi się w nim gośćmi oraz interakcja z nimi są w pełni zintegrowane z systemem operacyjnym za pomocą standardowych mechanizmów zarządzania, takich jak instrumentacja zarządzania Windows (WMI — ang. *Windows Management Instrumentation*) i usługi systemowe. W tym przypadku główny system operacyjny zawiera pewne rozszerzenia zwiększające „świadomość" maszyny wirtualnej. Rozszerzenia te to specjalne optymalizacje w jądrze i ewentualnie w sterownikach urządzeń, które wykrywają, że kod jest uruchamiany wirtualnie pod kontrolą hipernadzorcy, dzięki czemu pewne zadania są wykonywane inaczej lub bardziej wydajnie w tym środowisku.

Rysunek 9.1 przedstawia podstawową architekturę stosu wirtualizacji systemu Windows, która jest szczegółowo opisana w dalszej części tego rozdziału.



RYSUNEK 9.1. *Stos architektury platformy Hyper-V (hipernadzorca i stos wirtualizacji)*

W najniższej warstwie architektury znajduje się hipernadzorca, który jest uruchamiany bardzo wcześnie podczas startu systemu i udostępnia swoje usługi stosowi wirtualizacji (za pomocą interfejsu hiperwywołań (ang. *hypercall*)). Wczesna inicjalizacja hipernadzorcy jest opisana w rozdziale 12., „Uruchamianie i zamykanie systemu". Uruchomienie hipernadzorcy jest inicjalizowane przez program ładujący system Windows (ang. *loader*), który określa, czy uruchomić hipernadzorcę i **bezpieczne jądro** (ang. *secure kernel*); jeśli hipernadzorca i bezpieczne jądro zostaną uruchomione, hipernadzorca będzie korzystał z usług zapewnianych przez plik *Hvloader.dll* w celu wykrycia właściwej platformy sprzętowej oraz załadowania i uruchomienia odpowiedniej wersji hipernadzorcy. Ponieważ procesory Intel i AMD (oraz ARM64) różnią się implementacją wirtualizacji wspomaganej sprzętowo, istnieją różni hipernadzorcy. Właściwy z nich jest wybierany podczas uruchamiania systemu, po odpytaniu procesora za pomocą instrukcji CPUID. W systemach firmy Intel ładowany jest plik binarny

Hvix64.exe; w systemach firmy AMD używany jest obraz *Hvax64.exe*. Począwszy od aktualizacji systemu Windows 10 z maja 2019 roku (19H1), system Windows w wersji ARM64 obsługuje swojego własnego hipernadzorcę, który jest zaimplementowany w obrazie *Hvaa64.exe*.

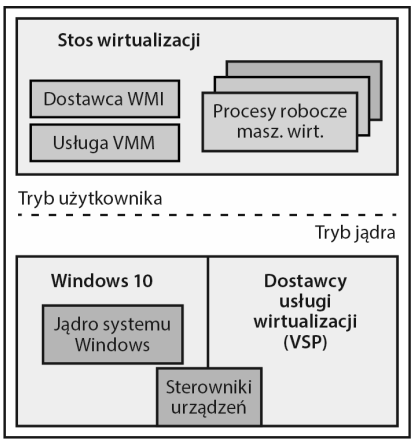
Na wysokim poziomie sprzętowe rozszerzenie wirtualizacji używane przez hipernadzorcę jest cienką warstwą, która znajduje się między jądrem systemu operacyjnego a procesorem. Warstwa ta, która przechwytuje i emuluje w bezpieczny sposób wrażliwe operacje wykonywane przez system operacyjny, jest uruchamiana na wyższym poziomie uprawnień niż jądro systemu operacyjnego. (Intel nazywa ten tryb VMXROOT, a w większości książek i publikacji domenę zabezpieczeń VMXROOT określa się jako „Pierścień-1" [ang. *Ring-1*]). Gdy operacja wykonywana przez bazowy system operacyjny zostanie przechwycona, procesor przerywa wykonywanie kodu systemu operacyjnego i przekazuje wykonanie do hipernadzorcy na wyższym poziomie przywilejów. Operacja ta jest powszechnie określana jako zdarzenie VMEXIT. W ten sam sposób, gdy hipernadzorca zakończy przetwarzanie przechwyconej operacji, potrzebuje sposobu, aby umożliwić fizycznemu procesorowi ponowne rozpoczęcie wykonywania kodu systemu operacyjnego. W rozszerzeniu wirtualizacji sprzętu zdefiniowano nowe kody operacyjne (ang. *opcodes*), które umożliwiają wykonanie zdarzenia VMENTER; procesor wznawia wykonywanie kodu systemu operacyjnego na pierwotnym poziomie uprawnień.

Partycje, procesy i wątki

Jednym z kluczowych komponentów architektonicznych stojących za hipernadzorcą Windows jest koncepcja partycji. Partycja zasadniczo odzwierciedla główną jednostkę izolowaną, wystąpienie, czyli instancję (ang. *instance*) instalacji systemu operacyjnego, która albo może odnosić się do tego, co tradycyjnie nazywa się hostem albo do tego co jest nazywane gościem. W modelu hipernadzorcy Windows nie używa się tych dwóch terminów; zamiast tego mówi się odpowiednio o partycji głównej (ang. *root partition*) lub partycji podrzędnej (ang. *child partition*). Partycja składa się z pewnej ilości pamięci fizycznej oraz jednego lub większej liczby procesorów wirtualnych (VP — ang. *Virtual Processor*) wraz z ich lokalnymi wirtualnymi Zaawansowanymi programowalnymi kontrolerami przerwań (APIC — ang. *Advanced Programmable Interrupt Controllers*) i czasomierzami. (W ujęciu globalnym partycja zawiera także wirtualną płytę główną i wiele wirtualnych urządzeń peryferyjnych. Są to koncepcje stosu wirtualizacji, które nie należą do hipernadzorcy).

Platforma Hyper-V posiada co najmniej partycję główną — na której jest uruchomiony główny system operacyjny sterujący maszyną — stos wirtualizacji i powiązane z nim komponenty. Każdy system operacyjny działający w zwirtualizowanym środowisku odzwierciedla partycję podrzędną, która może zawierać pewne dodatkowe narzędzia optymalizujące dostęp do sprzętu lub umożliwiające zarządzanie systemem operacyjnym. Partycje są zorganizowane w sposób hierarchiczny. Partycja główna sprawuje kontrolę nad każdą partycją podrzędną i otrzymuje powiadomienia (przechwytuje informacje) o pewnych zdarzeniach, które mają miejsce w partycji podrzędnej. Większość fizycznych dostępów do sprzętu, które są realizowane w partycji głównej, jest przekazywana za pomocą hipernadzorcy; oznacza to, że partycja nadrzędna może rozmawiać bezpośrednio ze sprzętem (z pewnymi wyjątkami). Z kolei partycje podrzędne zazwyczaj nie są w stanie komunikować się bezpośrednio ze sprzętem maszyny fizycznej (znów z pewnymi wyjątkami, które są opisane w dalszej części rozdziału, w podrozdziale „Stos wirtualizacji"). Każde wejście/wyjście jest przechwytywane przez hipernadzorcę i w razie potrzeby przekierowywane do partycji głównej (*root*).

Jednym z głównych założeń przy projektowaniu hipernadzorcy Windows było to, aby był on tak mały i modułowy, jak to tylko możliwe, podobnie jak mikrojądro — nie ma potrzeby obsługi żadnego sterownika hipernadzorcy ani dostarczania pełnego, monolitycznego modułu. Oznacza to, że większość działań związanych z wirtualizacją jest wykonywana przez oddzielny stos wirtualizacji (patrz rysunek 9.1). Hipernadzorca wykorzystuje istniejącą architekturę sterowników Windows i komunikuje się z rzeczywistymi sterownikami urządzeń Windows. Architektura ta składa się z kilku komponentów, które udostępniają to zachowanie i zarządzają nim, a które są wspólnie nazywane *stosem wirtualizacji*. Chociaż hipernadzorca jest odczytywany z dysku rozruchowego i wykonywany przez Windows Loader, zanim jeszcze zostanie uruchomiony główny system operacyjny (i zanim zostanie utworzona partycja nadrzędna), to właśnie partycja nadrzędna jest odpowiedzialna za obsługę całego stosu wirtualizacji. Ponieważ są to komponenty firmy Microsoft, na partycji głównej (*root*) może być uruchomiona tylko maszyna z systemem Windows. System operacyjny Windows na partycji głównej jest odpowiedzialny za dostarczanie sterowników urządzeń dla sprzętu zainstalowanego w systemie, a także za uruchamianie stosu wirtualizacji. Jest to również miejsce, w którym zarządzane są wszystkie partycje podrzędne. Główne komponenty partycji głównej pokazano na rysunku 9.2.

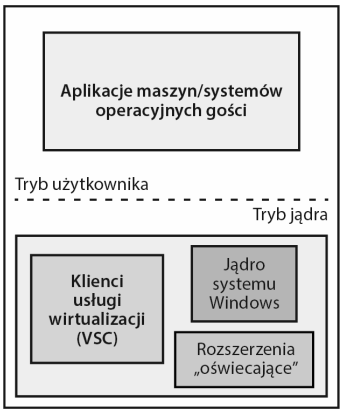


RYSUNEK 9.2. *Komponenty partycji głównej*

Partycje podrzędne

Partycja podrzędna jest to instancja dowolnego systemu operacyjnego działająca równolegle do partycji nadrzędnej. (Ponieważ można zapisać lub wstrzymać stan dowolnej partycji podrzędnej, niekoniecznie musi ona być uruchomiona). W odróżnieniu od partycji nadrzędnej, która ma pełny dostęp do APIC, portów I/O i swojej pamięci fizycznej (ale nie ma dostępu do pamięci fizycznej hipernadzorcy i bezpiecznego jądra), partycje podrzędne są ograniczone ze względów bezpieczeństwa i zarządzania do własnego widoku przestrzeni adresowej (fizycznej przestrzeni adresowej gości (GPA — ang. *Guest Physical Address*), która jest zarządzana przez hipernadzorcę). Partycje podrzędne nie mają również bezpośredniego dostępu do sprzętu (chociaż mogą mieć bezpośredni dostęp do niektórych urządzeń; więcej szczegółów w podrozdziale „Stos wirtualizacji"). Jeśli chodzi o dostęp do hipernadzorcy, partycja podrzędna jest również ograniczona głównie do otrzymywania i wysyłania powiadomień oraz do zmian stanu. Na przykład partycja podrzędna nie ma kontroli nad innymi partycjami (i nie może tworzyć nowych).

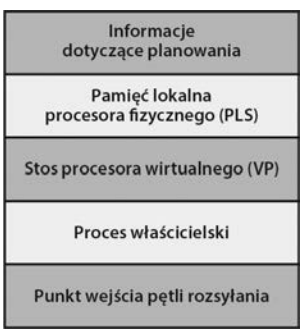
Partycje podrzędne mają znacznie mniej komponentów wirtualizacyjnych niż partycja nadrzędna, ponieważ nie są one odpowiedzialne za uruchamianie stosu wirtualizacji, a jedynie za komunikację z nim. Ponadto komponenty te można również uznać za opcjonalne, ponieważ zwiększają one wydajność środowiska, ale nie są krytyczne, jeśli chodzi o korzystanie z niego. Rysunek 9.3 przedstawia komponenty występujące w typowej partycji podrzędnej systemu Windows.



RYSUNEK 9.3. *Komponenty partycji podrzędnej*

Procesy i wątki

Hipernadzorca systemu Windows odwzorowuje maszynę wirtualną za pomocą struktury danych partycji. Partycja, jak opisano w poprzednim rozdziale, składa się z pewnej ilości pamięci (pamięci fizycznej gościa) i jednego lub większej liczby procesorów wirtualnych VP (ang. *Virtual Processors*). Wewnętrzny mechanizm zaimplementowany w hipernadzorcy działa w ten sposób, że każdy procesor wirtualny jest obiektem podlegającym planowaniu, a hipernadzorca, podobnie jak standardowe jądro NT, zawiera planistę (ang. *scheduler*). Planista przydziela każdemu fizycznemu procesorowi wykonywanie instrukcji wirtualnych procesorów, które należą do różnych partycji. (Różne typy planistów hipernadzorcy omówimy w dalszej części tego rozdziału w punkcie „Planiści platformy Hyper-V"). Wątek hipernadzorcy (struktura danych *TH_THREAD*) jest elementem łączącym wirtualny procesor z przypisaną do niego jednostką podlegającą planowaniu. Rysunek 9.4 przedstawia strukturę danych, która odwzorowuje bieżący fizyczny kontekst wykonania kodu. Zawiera ona stos wykonawczy wątku, dane dotyczące planowania, wskaźnik do wirtualnego procesora wątku, punkt wejścia pętli przydzielania wątku (omówiony później) oraz, co najważniejsze, wskaźnik do procesu hipernadzorcy, do którego należy wątek.



RYSUNEK 9.4. *Struktura danych wątku hipernadzorcy*

Hipernadzorca buduje wątek dla każdego tworzonego przez siebie procesora wirtualnego i kojarzy nowo stworzony wątek ze strukturą danych procesora wirtualnego (*VM_VP*).

Proces hipernadzorcy (struktura danych *TH_PROCESS*), pokazany na rysunku 9.5, odzwierciedla partycję i jest kontenerem dla jej fizycznej (i wirtualnej) przestrzeni adresowej. Proces ten zawiera listę wątków (które są obsługiwane przez wirtualne procesory), dane dotyczące planowania (powinowactwo fizycznych procesorów, w których proces ma prawo działać) oraz wskaźnik do podstawowych struktur danych pamięci partycji (przedział pamięci, zarezerwowane strony, główny katalog stron itd.). Proces jest zwykle tworzony, gdy hipernadzorca buduje partycję (*VM_PARTITION*), która będzie odzwierciedlać nową maszynę wirtualną.

RYSUNEK 9.5. *Struktura danych procesu hipernadzorcy*

Rozszerzenia zwiększające „świadomość" maszyny wirtualnej — „oświecenia"

Rozszerzenia zwiększające „świadomość" czyli innymi słowy „oświecające" maszynę wirtualną są jedną z kluczowych metod optymalizacji zwiększających wydajność, z których korzysta wirtualizacja systemu Windows. Są to bezpośrednie modyfikacje standardowego kodu jądra Windows, które potrafią wykryć, że system operacyjny jest uruchomiony na partycji podrzędnej i w rezultacie tego potrafią wykonać swoje zadania w inny sposób. Zazwyczaj te optymalizacje są w wysokim stopniu dostosowane do konkretnego sprzętu i powodują, że hiperwywołanie powiadamia hipernadzorcę.

Przykładem jest powiadamianie hipernadzorcy o długiej pętli oczekiwania z powodu zajętości (ang. *long busy-wait spin loop*). Hipernadzorca może zachować pewien stan na pętli zajętości i zdecydować o zaplanowaniu innego wirtualnego procesora na tym samym fizycznym procesorze, dopóki oczekiwanie z powodu zajętości nie będzie mogło zostać wypełnione. Wejście i wyjście ze stanu przerwania oraz dostęp do kontrolera APIC można skoordynować z hipernadzorcą, który może zostać „oświecony" po to, aby uniknąć pułapkowania rzeczywistego dostępu, a następnie jego wirtualizacji.

Inny przykład ma związek z zarządzaniem pamięcią, a konkretnie z opróżnianiem bufora TLB (ang. *Translation Lookaside Buffer* — bufor pośredni tłumaczenia). (Aby uzyskać więcej informacji na temat tych pojęć, patrz część I, rozdział 5., „Zarządzanie pamięcią"). Zazwyczaj system operacyjny wykonuje instrukcję procesora w celu „spłukania" jednego lub większej liczby nieaktualnych wpisów w buforze TLB, co ma wpływ tylko na jeden procesor. W systemach wieloprocesorowych zazwyczaj wpis w buforze TLB musi zostać wymieciony z pamięci podręcznej każdego aktywnego procesora (aby osiągnąć ten cel, system wysyła przerwanie międzyprocesorowe do każdego aktywnego procesora). Ponieważ jednak partycja podrzędna może współdzielić fizyczne procesory z wieloma innymi partycjami podrzędnymi, a niektóre z nich mogą wykonywać instrukcje wirtualnego procesora innej maszyny wirtualnej w momencie rozpoczęcia opróżniania bufora TLB, taka operacja spowodowałaby również „spłukanie" tej informacji w przypadku tych maszyn wirtualnych. Co więcej, wirtualny procesor zostałby ponownie skierowany do wykonania tylko przerwania międzyprocesorowego IPI (ang. *Interprocessor Interrupt*) czyszczącego bufor TLB, co spowodowałoby zauważalny spadek wydajności. Jeśli Windows działa pod kontrolą hipernadzorcy, w zastępstwie wysyła hiperwywołanie, po to aby hipernadzorca wyczyścił tylko określone informacje należące do partycji podrzędnej.

Przywileje partycji, właściwości i cechy wersji

Kiedy partycja jest na etapie początkowego tworzenia (zwykle przez sterownik wirtualizacji infrastruktury VID), nie są z nią związane żadne wirtualne procesory. W tym czasie sterownik VID może swobodnie dodawać lub usuwać niektóre przywileje partycji. W rzeczywistości, gdy partycja jest tworzona po raz pierwszy, hipernadzorca przypisuje jej pewne domyślne przywileje, w zależności od jej typu.

Przywilej partycji opisuje, jakie działania — zwykle wyrażone przez hiperwywołania lub syntetyczne rejestry MSR (ang. *Model Specific Registers*) dostosowane do konkretnego modelu — „oświecony" system operacyjny działający wewnątrz partycji może wykonywać w imieniu samej partycji. Na przykład przywilej **dostępu do planisty partycji głównej** (ang. *Access Root Scheduler*) pozwala partycji podrzędnej powiadomić partycję główną, że zdarzenie zostało zasygnalizowane i obsługa wirtualnego procesora systemu gościa może zostać przesunięta w czasie (to zwykle zwiększa priorytet wątku gościa obsługiwanego przez procesor wirtualny). Przywilej dostępu do **wirtualnego trybu bezpiecznego** (VSM — ang. *Virtual Secure Mode*) pozwala natomiast partycji włączyć wirtualny poziom zaufania (VTL — ang. *Virtual Trust Level*) 1 i uzyskać dostęp do jego właściwości i konfiguracji (zwykle eksponowanych poprzez rejestry syntetyczne). Tabela 9.1 wymienia wszystkie przywileje przypisane domyślnie przez hipernadzorcę.

TABELA 9.1. *Przywileje partycji*

TYP PARTYCJI	PRZYWILEJE DOMYŚLNE
Partycja główna i podrzędna	Odczyt/zapis licznika czasu pracy procesora wirtualnego
	Odczyt bieżącego czasu referencyjnego partycji
	Dostęp do czasomierzy i rejestrów syntetycznego kontrolera przerwań SynIC
	Odczyt/ustawienie wirtualnej strony pomocy kontrolera APIC procesora wirtualnego
	Odczyt/zapis hiperwywołania programów obsługi pamięci (MSR — ang. *Memory Service Routines*)
	Żądanie dokonania wpisu IDLE w procesorze wirtualnym
	Odczyt indeksu procesora wirtualnego
	Mapowanie lub usuwanie mapowania obszaru kodowego hiperwywołania
	Odczyt emulowanego licznika znaczników czasu (TSC — ang. *time-stamp counter*) procesora wirtualnego i częstotliwości taktowania procesora
	Kontrola emulacji licznika znaczników czasu TSC partycji i emulacji ponownego „oświecenia" maszyny wirtualnej
	Odczyt/zapis rejestrów syntetycznych wirtualnego trybu bezpiecznego (VSM — ang. *Virtual Secure Mode*)
	Odczyt/zapis rejestrów procesora wirtualnego za pośrednictwem wirtualnych poziomów zaufania (VTL)
	Uruchomienie wirtualnego procesora aplikacji (AP — ang. *Application Processor*)
	Włączenie obsługi szybkich hiperwywołań partycji

TABELA 9.1. *Przywileje partycji — ciąg dalszy*

TYP PARTYCJI	PRZYWILEJE DOMYŚLNE
Tylko partycja główna	Tworzenie partycji podrzędnej
	Wyszukiwanie i odwoływanie się do partycji po jej ID
	Deponowanie/wycofywanie pamięci z przedziału partycji
	Wysyłanie wiadomości do portu podłączenia
	Sygnalizowanie zdarzenia w partycji portu podłączenia
	Tworzenie/usuwanie i uzyskiwanie właściwości portu podłączenia partycji
	Połączenie/rozłączenie się z portem podłączenia partycji
	Mapowanie/usuwanie mapowania strony statystyk hipernadzorcy (opisujących procesor wirtualny, procesor logiczny, partycję lub hipernadzorcę)
	Włączenie debugera hipernadzorcy dla partycji
	Planowanie pracy procesora wirtualnego partycji podrzędnej i uzyskanie dostępu do syntetycznych programów obsługi pamięci MSR syntetycznego kontrolera przerwań SynIC
	Wyzwolenie resetu systemu „oświeconego”
	Odczytanie opcji debugera hipernadzorcy dla partycji
Tylko partycja podrzędna	Generowanie przechwytu rozszerzonego hiperwywołania w partycji głównej
	Powiadomienie wątku obsługiwanego przez procesor wirtualny planisty głównego (*root*) o sygnalizowanym zdarzeniu
Partycja EXO	Brak

Przywileje partycji mogą być ustawione tylko przed utworzeniem i uruchomieniem przez partycję dowolnego procesora wirtualnego; hipernadzorca nie pozwoli na żądanie ustawienia przywilejów po rozpoczęciu wykonywania pojedynczego procesora wirtualnego w partycji. Właściwości partycji są podobne do przywilejów, ale nie mają tego ograniczenia; mogą być ustawiane i odpytywane w dowolnym momencie. Istnieją różne grupy właściwości, które można odszukać lub ustawić dla partycji. Tabela 9.2 zawiera listę grup właściwości.

TABELA 9.2. *Właściwości partycji*

GRUPA WŁAŚCIWOŚCI	OPIS
Właściwości dotyczące planowania	Ustawianie/wyszukiwanie właściwości związanych z planistą klasycznym i planistą rdzenia, takich jak **Limit** (*Cap*), **Waga** (*Weight*) i **Rezerwa** (*Reserve*)
Właściwości czasu	Zezwalanie na zawieszenie/wznowienie partycji
Właściwości debugowania	Zmiana konfiguracji uruchomieniowej debugera hipernadzorcy
Właściwości zasobów	Zapytanie o właściwości specyficzne dla platformy sprzętu wirtualnego (takie jak rozmiar TLB, obsługa rozszerzenia SGX (ang. *Software Guard Extension*) itd.)
Właściwości zgodności (kompatybilności)	Zapytania o właściwości specyficzne dla platformy sprzętu wirtualnego, które są związane z początkowymi funkcjami kompatybilności

Gdy tworzona jest partycja, sterownik wirtualizacji infrastruktury VID zapewnia hipernadzorcy poziom kompatybilności (który jest określony w pliku konfiguracyjnym maszyny wirtualnej). Na podstawie tego poziomu zgodności hipernadzorca włącza lub wyłącza określone funkcje sprzętu wirtualnego, które mogłyby zostać zaprezentowane przez procesor wirtualny bazowemu systemowi operacyjnemu. Istnieje wiele funkcji, które dostosowują zachowanie procesora wirtualnego w zależności od poziomu kompatybilności maszyny wirtualnej. Dobrym przykładem może być sprzętowa tablica atrybutów stron (PAT — ang. *Page Attribute Table*), która jest konfigurowalnym typem buforowania pamięci wirtualnej. Przed aktualizacją rocznicową systemu Windows 10 (RS1) w maszynach wirtualnych gości nie można było używać PAT, więc niezależnie od tego, czy poziom zgodności maszyny wirtualnej określa Windows 10 RS1, hipernadzorca nie zaprezentuje rejestrów PAT bazowemu systemowi operacyjnemu gościa. W przeciwnym razie, w przypadku gdy poziom zgodności jest wyższy niż Windows 10 RS1, hipernadzorca prezentuje obsługę PAT bazowemu systemowi operacyjnemu uruchomionemu w maszynie wirtualnej gościa. Gdy partycja główna jest początkowo tworzona w czasie rozruchu, hipernadzorca włącza dla niej najwyższy poziom zgodności. W ten sposób główny system operacyjny (*root*) może korzystać ze wszystkich funkcji obsługiwanych przez sprzęt fizyczny.

Uruchamianie hipernadzorcy

W rozdziale 12. analizujemy sposób uruchamiania stacji roboczej opartej na UEFI oraz wszystkie komponenty zaangażowane w załadowanie i uruchomienie właściwej wersji binarnej hipernadzorcy. W tym rozdziale krótko omawiamy, co się dzieje w maszynie po tym, jak moduł ładujący hipernadzorcę HvLoader przekazał wykonanie do hipernadzorcy, który po raz pierwszy przejmuje kontrolę.

HvLoader ładuje odpowiednią wersję obrazu binarnego hipernadzorcy (w zależności od producenta procesora) i tworzy blok ładujący hipernadzorcy. Przechwytuje on minimalny kontekst procesora, który jest potrzebny hipernadzorcy do uruchomienia pierwszego wirtualnego procesora. HvLoader przełącza się następnie na nową, dopiero co utworzoną przestrzeń adresową i przenosi wykonanie do obrazu hipernadzorcy poprzez wywołanie punktu wejścia obrazu hipernadzorcy, *KiSystemStartup*, który przygotowuje procesor do uruchomienia hipernadzorcy i inicjalizuje strukturę danych *CPU_PLS*. *CPU_PLS* odwzorowuje fizyczny procesor i działa jak struktura danych bloku sterującego procesora PRCB jądra NT; hipernadzorca jest w stanie szybko ją zaadresować (używając segmentu GS). W odróżnieniu od jądra NT *KiSystemStartup* jest wywoływany tylko dla procesora startowego (sekwencja uruchamiania procesorów aplikacji jest omówiona w punkcie „Uruchamianie procesorów aplikacji" w dalszej części tego rozdziału), dlatego odracza prawdziwą inicjalizację na rzecz wykonania innej funkcji, *BmpInitBootProcessor*.

BmpInitBootProcessor rozpoczyna złożoną sekwencję inicjalizacji. Funkcja bada system i odpytuje wszystkie obsługiwane przez CPU funkcje wirtualizacji (takie jak EPT i VPID; odpytane funkcje są specyficzne dla platformy i różnią się w zależności od wersji hipernadzorcy Intela, AMD lub ARM). Następnie wyznacza planistę hipernadzorcy, który to planista będzie zarządzał sposobem, w jaki hipernadzorca będzie planował pracę wirtualnych procesorów. Dla systemów serwerowych Intel i AMD domyślnym planistą jest planista rdzenia (ang. *core scheduler*), podczas gdy planista partycji głównej (ang. *root scheduler*) jest domyślny dla wszystkich systemów klienckich (w tym ARM64). Typ planisty może być ręcznie nadpisany poprzez opcję BCD *hypervisorschedulertype* (więcej informacji o różnych planistach hipernadzorcy jest dostępnych w dalszej części tego rozdziału).

Inicjalizowane są zagnieżdżone rozszerzenia „oświecające". Zagnieżdżone rozszerzenia „oświecające" pozwalają hipernadzorcy na wykonywanie instrukcji w zagnieżdżonych konfiguracjach, gdzie hipernadzorca główny (zwany hipernadzorcą L0), zarządza rzeczywistym sprzętem, a inny hipernadzorca (zwany hipernadzorcą L1) jest wykonywany w maszynie wirtualnej. Po tym etapie program *BmpInitBootProcessor* przeprowadza inicjalizację komponentów takich jak:

- Menedżer pamięci (inicjalizuje bazę numerów ramek stron (PFN — ang. *Page Frame Number*) i przedział główny).
- Warstwa abstrakcji sprzętowej HAL hipernadzorcy (HAL — ang. *Hardware Abstraction Layer*).
- Podsystem procesów i wątków hipernadzorcy (który zależy od wybranego typu planisty). Tworzony jest proces systemowy i jego początkowy wątek. Ten proces jest specjalny, nie jest związany z żadną partycją i jest gospodarzem/hostem wątków wykonujących kod hipernadzorcy.
- Warstwa abstrakcji wirtualizacji (VAL — ang. *Virtualization Abstraction Layer*) trybu VMX. Zadaniem warstwy VAL jest abstrakcja różnic pomiędzy wszystkimi obsługiwanymi rozszerzeniami wirtualizacji sprzętu (Intel, AMD i ARM64). Warstwa ta zawiera kod, który operuje na specyficznych dla danej platformy cechach technologii wirtualizacji maszyny, wykorzystywanych przez hipernadzorcę (na przykład na platformie Intela warstwa VAL zarządza obsługą „nieograniczonego gościa", EPT, SGX, MBEC i tak dalej).
- Syntetyczny kontroler przerwań (SynIC) i jednostka zarządzania pamięcią I/O (IOMMU).
- Menedżer adresów (AM — ang. *Address Manager*), który jest komponentem odpowiedzialnym za zarządzanie pamięcią fizyczną przypisaną do partycji (zwaną pamięcią fizyczną gościa lub GPA (ang. *Guest Physical Memory*)) i jej translację do rzeczywistej pamięci fizycznej (zwanej systemową pamięcią fizyczną). Choć pierwsza implementacja platformy Hyper-V obsługiwała tablice pseudo-stron (ang. *shadow page tables*) mapujące odwołania gościa do adresów fizycznych (jest to programowa technika translacji adresów), począwszy od Windows 8.1 menedżer adresów wykorzystuje zależny od platformy kod do konfiguracji mechanizmu translacji adresów hipernadzorcy, który to mechanizm jest oferowany przez warstwę sprzętową (rozszerzone tablice stron w przypadku Intela, tablice stron zagnieżdżonych w przypadku AMD). W terminologii hipernadzorcy fizyczna przestrzeń adresowa partycji jest nazywana domeną adresową. Niezależna od platformy translacja fizycznej przestrzeni adresowej jest powszechnie nazywana translacją adresów warstwy drugiej (SLAT — ang. *Second Layer Address Translation*). Termin ten odnosi się do dwustopniowego mechanizmu translacji adresów EPT firmy Intel, NPT firmy AMD lub ARM.

Hipernadzorca może teraz zakończyć konstruowanie struktury danych *CPU_PLS* związanej z procesorem rozruchowym poprzez przydzielenie początkowych, zależnych od sprzętu struktur sterujących maszyny wirtualnej (VMCS w przypadku Intela, VMCB w przypadku AMD) oraz poprzez włączenie wirtualizacji poprzez pierwszą operację VMXON. Na koniec inicjalizowane są struktury danych mapowania przerwań dla poszczególnych procesorów.

Eksperyment: podłączenie debugera hipernadzorcy

W tym eksperymencie podłączysz debuger hipernadzorcy w celu przeanalizowania sekwencji startowej hipernadzorcy, co zostało omówione w poprzednim rozdziale. Debuger hipernadzorcy jest obsługiwany tylko przez interfejs przewodowy szeregowy lub sieciowy. Do debugowania hipernadzorcy można używać tylko maszyn fizycznych lub maszyn wirtualnych, w których włączona jest funkcja „wirtualizacji zagnieżdżonej" (patrz punkt „Wirtualizacja zagnieżdżona" w dalszej części tego rozdziału). W tym ostatnim przypadku tylko debugowanie przez interfejs szeregowy może być włączone dla wirtualizowanego hipernadzorcy L1.

Do tego eksperymentu potrzebna jest osobna maszyna fizyczna, która obsługuje rozszerzenia wirtualizacji oraz ma zainstalowaną i włączoną rolę Hyper-V. Będziesz używał tej maszyny jako systemu debugowanego, podłączonego do systemu hosta (który działa jako debuger), gdzie uruchamiasz narzędzia do debugowania. Alternatywnie możesz skonfigurować zagnieżdżoną maszynę wirtualną, jak pokazano w eksperymencie „Włączanie zagnieżdżonej wirtualizacji w platformie Hyper-V" w dalszej części tego rozdziału (w tym przypadku nie potrzebujesz kolejnej maszyny fizycznej).

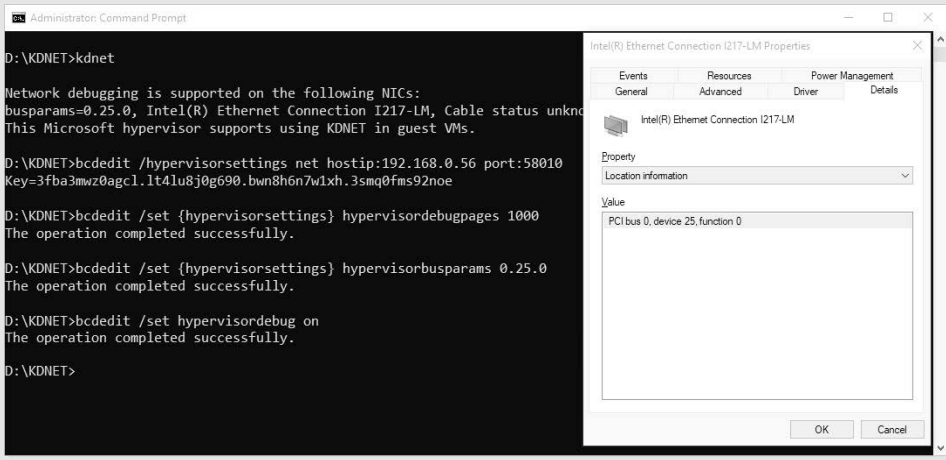
W pierwszym kroku pobierz i zainstaluj w systemie hosta aplikację Debugging Tools for Windows (Narzędzia do debugowania dla Windows), która jest dostępna jako część Windows SDK (lub WDK), do pobrania ze strony *https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk*. Alternatywnie do tego eksperymentu można również użyć WinDbgX, który w chwili pisania tego tekstu jest dostępny w Windows Store poprzez wyszukiwanie `WinDbg Preview`.

Debugowany system do tego eksperymentu musi mieć wyłączoną opcję **bezpiecznego rozruchu** (ang. *Secure Boot*). Debugowanie hipernadzorcy nie jest kompatybilne z bezpiecznym rozruchem. Zapoznaj się z instrukcją obsługi stacji roboczej, aby dowiedzieć się, jak wyłączyć bezpieczny rozruch (zazwyczaj ustawienia bezpiecznego rozruchu znajdują się w UEFI BIOS). Aby włączyć debuger hipernadzorcy w debugowanym systemie, najpierw otwórz administracyjny wiersz poleceń (wpisz **`cmd`** w polu wyszukiwania na pasku zadań, a następnie kliknij prawym przyciskiem myszy ikonę *Command Prompt* (*Wiersz polecenia*) i wybierz opcję *Run as administrator* (*Uruchom jako administrator*)).

W przypadku gdy chcesz debugować hipernadzorcę poprzez kartę sieciową, powinieneś wpisać następujące polecenia, zastępując wyrażenia <*IpHosta*> adresem IP systemu hosta; <*PortHosta*> prawidłowym portem w hoście (od portu 49152); oraz <*ParamMagistrKartSiec*> parametrami magistrali karty sieciowej debugowanego systemu, określonymi w formacie *XX.YY.ZZ* (gdzie *XX* to numer magistrali, *YY* to numer urządzenia, a *ZZ* to numer funkcji). Parametry magistrali karty sieciowej można odkryć poprzez aplet Device Manager lub poprzez narzędzie KDNET.exe dostępne w Windows SDK:

```
bcdedit /hypervisorsettings net hostip:<IpHosta> port:<PortHosta>
bcdedit /set {hypervisorsettings} hypervisordebugpages 1000
bcdedit /set {hypervisorsettings} hypervisorbusparams <ParamMagistrKartSiec>
bcdedit /set hypervisordebug on
```

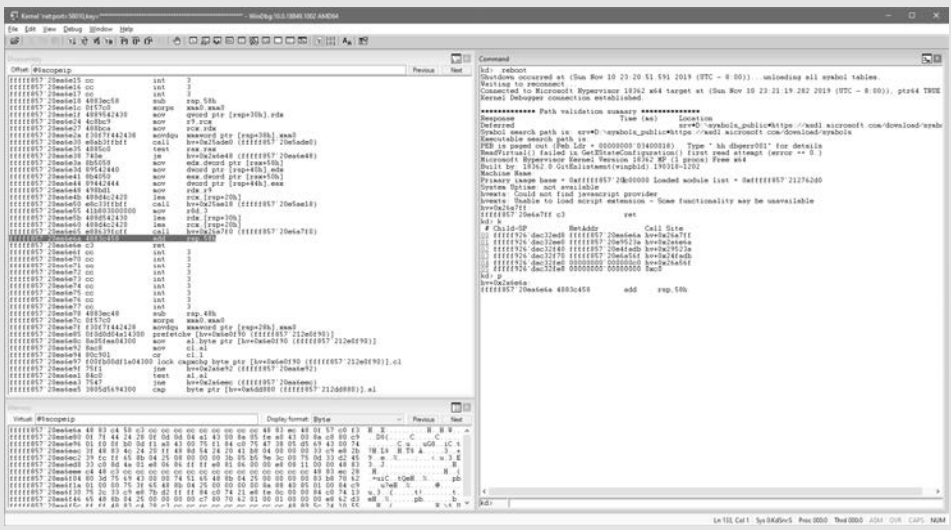
Poniższy rysunek przedstawia przykładowy system, w którym interfejs sieciowy używany do debugowania hipernadzorcy znajduje się w parametrach magistrali 0.25.0, a debuger jest skierowany do systemu hosta skonfigurowanego z adresem IP 192.168.0.56 na porcie 58010.



Zwróć uwagę na zwrócony klucz debugowania. Po ponownym uruchomieniu debugowanego systemu uruchom w hoście Windbg, wydając następujące polecenie:

```
windbg.exe -d -k net:port=<PortHosta>,key=<KluczDebugowania>
```

Debugowanie hipernadzorcy i śledzenie sekwencji jego uruchamiania powinno być możliwe, nawet jeśli Microsoft nie udostępni symboli głównego modułu hipernadzorcy:



W maszynie wirtualnej z włączoną wirtualizacją zagnieżdżoną można włączyć debuger hipernadzorcy L1 tylko przez port szeregowy, stosując w debugowanym systemie następujące polecenie:

```
bcdedit /hypervisorsettings SERIAL DEBUGPORT:1 BAUDRATE:115200
```

Tworzenie partycji głównej i wirtualnego procesora rozruchowego

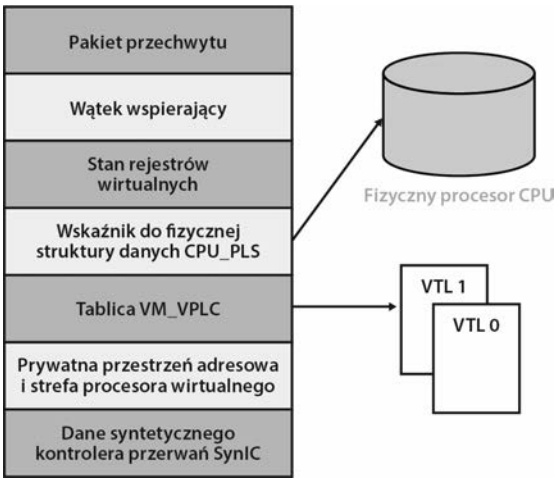
Pierwszymi krokami, które musi wykonać w pełni zainicjalizowany hipernadzorca, jest utworzenie partycji głównej i pierwszego procesora wirtualnego używanego do uruchamiania systemu (nazywanego BSP VP). Tworzenie partycji głównej odbywa się według prawie takich samych zasad jak w przypadku partycji podrzędnych; wiele warstw partycji jest inicjalizowanych jedna po drugiej. W szczególności:

1. Warstwa maszyny wirtualnej VM inicjalizuje maksymalną dozwoloną liczbę poziomów VTL-a i ustawia uprawnienia partycji w oparciu o jej typ (więcej szczegółów w poprzedniej sekcji). Ponadto warstwa VM określa dopuszczalne właściwości partycji na podstawie określonego poziomu zgodności partycji. Partycja główna obsługuje maksymalne dopuszczalne funkcje.
2. Warstwa procesora wirtualnego inicjalizuje zwirtualizowane dane identyfikatora głównego procesora (CPUID — ang. *Central Processing Unit Identifier*), których używają wszystkie wirtualne procesory partycji, gdy CPUID jest żądany od systemu operacyjnego gościa. Warstwa procesora wirtualnego tworzy proces hipernadzorcy, który obsługuje partycję.
3. Menedżer adresów (AM) konstruuje początkową fizyczną przestrzeń adresową partycji przy użyciu kodu zależnego od platformy maszynowej (który buduje EPT w przypadku Intela, NPT w przypadku AMD). Konstruowana fizyczna przestrzeń adresowa zależy od typu partycji. Partycja główna (*root*) wykorzystuje mapowanie tożsamości, co oznacza, że cała pamięć fizyczna gościa odpowiada systemowej pamięci fizycznej (więcej informacji na ten temat znajduje się w dalszej części rozdziału w punkcie „Fizyczna przestrzeń adresowa partycji").

Wreszcie po tym jak SynIC, IOMMU i strony współdzielone przechwytów są poprawnie skonfigurowane dla partycji, hipernadzorca tworzy i uruchamia wirtualny procesor BSP dla partycji głównej, który jest unikalnym procesorem używanym do ponownego uruchomienia procesu startowego.

Procesor wirtualny hipernadzorcy jest odzwierciedlany przez dużą strukturę danych (*VM_VP*), pokazaną na rysunku 9.6. Struktura danych maszyny wirtualnej i procesora wirtualnego VM_VP utrzymuje wszystkie dane używane do śledzenia stanu wirtualnego procesora: stan jego rejestrów zależnych od platformy (takich jak ogólnego przeznaczenia, debugowania, obszaru XSAVE i stosu) i danych, prywatną przestrzeń adresową procesora wirtualnego oraz tablicę struktur danych *VM_VPLC*, które są używane do śledzenia stanu każdego wirtualnego poziomu zaufania (VTL) procesora wirtualnego. *VM_VP* zawiera również wskaźnik do wątku wspierającego procesora wirtualnego oraz wskaźnik do procesora fizycznego, który aktualnie wykonuje instrukcje procesora wirtualnego.

Podobnie jak w przypadku partycji, tworzenie wirtualnego procesora BSP jest podobne do procesu tworzenia zwykłych wirtualnych procesorów. *VmAllocateVp* jest funkcją odpowiedzialną za przydzielenie i inicjalizację potrzebnej pamięci z przedziału partycji, używanej do przechowywania struktury danych *VM_VP*, jej części zależnej od platformy oraz tablicy *VM_VPLC* (po jednej dla każdego obsługiwanego VTL-a). Hipernadzorca kopiuje do struktury *VM_VP* początkowy kontekst procesora, określony przez HvLoader podczas startu systemu, a następnie tworzy prywatną przestrzeń adresową procesora wirtualnego i dołącza się do niej (tylko w przypadku, gdy włączona jest izolacja przestrzeni adresowej). Na koniec tworzy wątek wspierający procesora wirtualnego. Jest to ważny krok: budowa wirtualnego procesora trwa w kontekście jego własnego wątku wspierającego. Główny wątek systemowy hipernadzorcy na tym etapie czeka, aż nowy procesor wirtualny używany do uruchamiania systemu BSP VP zostanie całkowicie zainicjalizowany. Czekanie powoduje, że planista hipernadzorcy wybiera nowo utworzony wątek, który wykonuje program, *ObConstructVp*, konstruujący procesor wirtualny w kontekście nowego wątku wspieranego.



RYSUNEK 9.6. *Struktura danych VM_VP odzwierciedlająca procesor wirtualny*

ObConstructVp, w podobny sposób jak w przypadku partycji, konstruuje i inicjalizuje każdą warstwę procesora wirtualnego — w szczególności:

1. Warstwa menedżera wirtualizacji (VM — ang. *Virtualization Manager*) dołącza strukturę danych procesora fizycznego (*CPU_PLS*) do procesora wirtualnego i ustawia VTL 0 jako aktywny.
2. Warstwa VAL inicjalizuje zależne od platformy części procesora wirtualnego, takie jak jego rejestry, obszar XSAVE, stos i dane debugowania. Ponadto dla każdego obsługiwanego VTL-a, warstwa VAL przydziela i inicjalizuje strukturę danych VMCS (VMCB w przypadku systemów AMD), która jest używana przez sprzęt do śledzenia stanu maszyny wirtualnej, oraz tablice stron SLAT VTL-a. Te ostatnie pozwalają na odizolowanie każdego wirtualnego poziomu zaufania (VTL) od innych (więcej szczegółów na temat wirtualnych poziomów zaufania znajduje się w dalszej części rozdziału, w punkcie „Wirtualne poziomy zaufania (VTL) i wirtualny tryb bezpieczny (VSM)"). Na koniec warstwa VAL włącza i ustawia VTL 0 jako aktywny. Zależne od platformy struktury sterujące maszyną wirtualną (VMCS — ang. *Virtual Machine Control Structure* — lub VMCB dla systemów AMD) są całkowicie kompilowane, tablica SLAT dla VTL 0 jest ustawiana jako aktywna, a emulator trybu rzeczywistego jest inicjalizowany. Część struktur VMCS, dotycząca stanu hosta, jest kierowana na pętlę dyspozycyjną VAL hipernadzorcy. Ten program jest najważniejszą częścią hipernadzorcy, ponieważ zarządza wszystkimi zdarzeniami VMEXIT generowanymi przez każdego gościa.
3. Warstwa procesora wirtualnego przydziela stronę hiperwywołania procesora wirtualnego, a także, dla każdego wirtualnego poziomu zaufania (VTL), strony komunikatów pomocy (*assist*) i przechwytów (*intercept*). Strony te są wykorzystywane przez hipernadzorcę do udostępniania kodu lub danych systemowi operacyjnemu gościa.

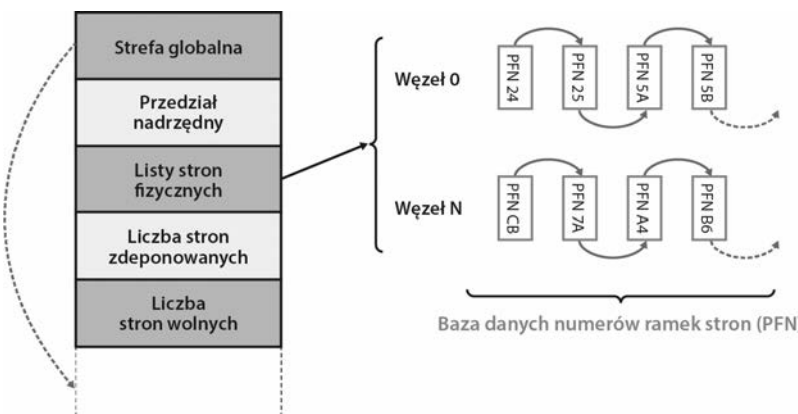
Gdy *ObConstructVp* zakończy swoją pracę, wątek rozsyłania procesora wirtualnego aktywuje procesor wirtualny i jego syntetyczny kontroler przerwań (SynIC). Jeśli procesor wirtualny jest pierwszy na partycji głównej, wątek rozsyłania przywraca początkowy kontekst procesora wirtualnego przechowywany w strukturze danych *VM_VP*, zapisując każdy przechwycony rejestr w obszarze procesora obsługującym zależne od platformy Struktury sterujące maszyną

wirtualną VMCS (lub VMCB) (kontekst został określony przez HvLoader wcześniej w procesie startowym). Wątek rozsyłania sygnalizuje w końcu zakończenie inicjalizacji procesora wirtualnego (w efekcie główny wątek systemowy wchodzi do pętli bezczynności) i wchodzi do zależnej od platformy pętli rozsyłania VAL. Pętla rozsyłania VAL wykrywa, że procesor wirtualny jest nowy, przygotowuje go do pierwszego wykonania i uruchamia nową maszynę wirtualną, wykonując instrukcję VMLAUNCH. Nowa maszyna wirtualna uruchamia się ponownie dokładnie w punkcie, w którym HvLoader przekazał wykonanie do hipernadzorcy. Proces uruchamiania jest kontynuowany normalnie, ale w kontekście nowej partycji hipernadzorcy.

Menedżer pamięci hipernadzorcy

Menedżer pamięci hipernadzorcy jest stosunkowo prosty w porównaniu z menedżerem pamięci dla NT czy bezpiecznego jądra. Podmiotem zarządzającym zbiorem stron pamięci fizycznej jest *przedział pamięci* hipernadzorcy. Zanim nastąpi start hipernadzorcy, program ładujący hipernadzorcę (*Hvloader.dll*) przydziela blok ładujący hipernadzorcę i wstępnie oblicza maksymalną liczbę stron fizycznych, które zostaną wykorzystane przez hipernadzorcę do prawidłowego uruchomienia i utworzenia partycji głównej. Liczba ta zależy od stron używanych do inicjalizacji IOMMU w celu przechowywania struktur zakresu pamięci, systemowej bazy PFN, tablic stron SLAT oraz wirtualnej przestrzeni adresowej VA warstwy HAL. Program ładujący hipernadzorcę wstępnie przydziela obliczoną liczbę stron fizycznych, oznacza je jako zarezerwowane i dołącza tablicę listy stron w bloku ładującym. Później, gdy hipernadzorca startuje, tworzy przedział główny, korzystając z listy stron, która została przydzielona przez program ładujący hipernadzorcę.

Rysunek 9.7 przedstawia układ struktury danych przedziału pamięci. Struktura danych śledzi całkowitą liczbę stron fizycznych „zdeponowanych" w przedziale, które mogą zostać gdzieś przydzielone lub uwolnione. Przedział przechowuje swoje strony fizyczne w różnych listach uporządkowanych według węzła Niejednolitego dostępu do pamięci NUMA (ang. *Non-uniform memory access*). Tylko nagłówek każdej listy jest przechowywany w przedziale. Stan każdej strony fizycznej i jej powiązanie na liście NUMA jest utrzymywane dzięki wpisom w bazie danych PFN. Przedział śledzi również swoje powiązanie z partycją główną (*root*). Nowy przedział może zostać utworzony przy użyciu stron fizycznych należących do partycji nadrzędnej (*root*). Podobnie, gdy przedział jest usuwany, wszystkie jego pozostałe strony fizyczne są zwracane do partycji nadrzędnej.



RYSUNEK 9.7. *Przedział pamięci hipernadzorcy. Wirtualna przestrzeń adresowa dla strefy globalnej jest rezerwowana od końca struktury danych przedziału*

Kiedy hipernadzorca potrzebuje trochę pamięci fizycznej do jakiejkolwiek pracy, przydziela z aktywnego przedziału (w zależności od partycji). Oznacza to, że przydzielenie może się nie powieść. W przypadku niepowodzenia mogą się pojawić dwa możliwe scenariusze:

- Jeśli przydzielenie zostało zażądane dla usługi wewnętrznej dla hipernadzorcy (zwykle w imieniu partycji głównej), awaria nie powinna mieć miejsca, a system jednak ulega awarii. (To wyjaśnia, dlaczego początkowe obliczenie całkowitej liczby stron, które mają być przydzielone do przedziału głównego, musi być dokładne).
- Jeśli przydzielenie zostało zażądane w imieniu partycji podrzędnej (zwykle poprzez hiperwywołanie), hipernadzorcy nie uda się wysłać żądania ze statusem *INSUFFICIENT_MEMORY*. Partycja główna wykrywa błąd i wykonuje przydzielenie pewnej strony fizycznej (więcej szczegółów omówiono w dalszej części rozdziału, w podrozdziale „Stos wirtualizacji"), która zostanie zdeponowana w przedziale podrzędnym poprzez hiperwywołanie *HvDepositMemory*. Operacja ta może być w końcu ponownie zainicjalizowana (i zazwyczaj zakończy się powodzeniem).

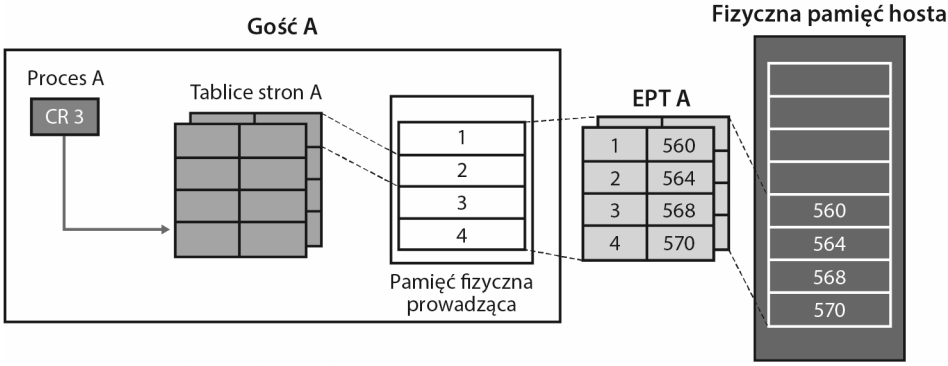
Fizyczne strony przydzielone z przedziału są zwykle mapowane w hipernadzorcy za pomocą adresu wirtualnego. Gdy tworzony jest przedział, przydzielany jest zakres adresów wirtualnych (o wielkości 4 lub 8 GB, w zależności od tego, czy przedział jest główny (*root*) czy podrzędny (*child*)) w celu zmapowania nowego przedziału, jego bitmapy PDE i jego globalnej strefy.

Strefa hipernadzorcy enkapsuluje prywatny zakres adresów wirtualnych VA, który nie jest współdzielony z całą przestrzenią adresową hipernadzorcy (patrz punkt „Izolacja przestrzeni adresowej" w dalszej części rozdziału). Hipernadzorca wykonuje instrukcje z pojedynczą tablicą stron głównych (inaczej niż jądro NT, które używa techniki tworzenia pseudo-adresów wirtualnych jądra (ang. *Kernel Virtual Address shadowing*)). Dwa wpisy w na stronie tablicy stron głównych są zarezerwowane z myślą o dynamicznym przełączaniu się pomiędzy każdą ze stref a przestrzeniami adresowymi procesorów wirtualnych.

Fizyczna przestrzeń adresowa partycji

Jak zostało to omówione w poprzednim rozdziale, kiedy partycja jest początkowo tworzona, hipernadzorca przydziela dla niej fizyczną przestrzeń adresową. Fizyczna przestrzeń adresowa zawiera wszystkie struktury danych potrzebne sprzętowi do translacji adresów fizycznych gości (GPA — ang. *Guest Physical Address*) partycji na systemowe adresy fizyczne (SPA — ang. *System Physical Address*). Funkcja sprzętowa umożliwiająca translację jest ogólnie określana jako translacja adresu drugiego poziomu (SLAT — ang. *Second Level Address Translation*). Termin *SLAT* jest uniwersalny (niezależny od platformy) — producenci sprzętu używają różnych nazw: Intel nazywa to rozwiązanie rozszerzonymi tablicami stron (EPT — ang. *Extended Page Tables*), AMD używa terminu tablice stron zagnieżdżonych (NPT — ang. *Nested Page Tables*), a ARM po prostu nazywa to translacją adresów etapu 2. (ang. *Stage 2 Address Translation*).

SLAT jest zwykle implementowany w sposób podobny do implementacji tablic stron architektury x64, która wykorzystuje cztery poziomy translacji (translacja adresów wirtualnych x64 została już szczegółowo omówiona w rozdziale 5. części I). System operacyjny uruchomiony wewnątrz partycji używa tej samej translacji adresów wirtualnych, jak gdyby był uruchamiany przez sprzęt bare-metalowy. Jednak w tym pierwszym przypadku procesor fizyczny faktycznie wykonuje dwa poziomy translacji: jeden dla adresów wirtualnych i jeden dla tłumaczenia adresów fizycznych. Rysunek 9.8 przedstawia konfigurację SLAT dla partycji gościa. W partycji gościa GPA jest zwykle tłumaczony na różniący się od niego SPA. Nie dzieje się tak w przypadku partycji głównej.



RYSUNEK 9.8. *Translacja adresów dla partycji gościa*

Kiedy hipernadzorca tworzy partycję główną, buduje jej początkową fizyczną przestrzeń adresową za pomocą mapowania tożsamości. W tym modelu każdy GPA odpowiada temu samemu SPA (na przykład ramka gościa 0x1000 w partycji głównej jest mapowana do fizycznej ramki bare-metalowej 0x1000). Hipernadzorca wstępnie przydziela pamięć potrzebną do zmapowania całej fizycznej przestrzeni adresowej maszyny (która została wykryta przez program ładujący Windows za pomocą usług UEFI; szczegóły w rozdziale 12.) na wszystkie dozwolone wirtualne poziomy zaufania (VTL) partycji głównej. (Partycja główna obsługuje zwykle dwa VTL-e). Tablice stron SLAT każdego VTL-a należącego do partycji zawierają te same wpisy GPA i SPA, ale zwykle z ustawionym innym poziomem ochrony. Poziom ochrony zastosowany do fizycznej ramki każdej partycji umożliwia tworzenie różnych domen zabezpieczeń, które mogą być izolowane jedna od drugiej. VTL-e są szczegółowo wyjaśnione w punkcie „Bezpieczne jądro" w dalszej części tego rozdziału. Strony hipernadzorcy są oznaczone jako sprzętowo zarezerwowane i nie są mapowane w tablicy SLAT partycji (w rzeczywistości są mapowane przy użyciu nieprawidłowego wpisu wskazującego na fikcyjny numer ramki strony (PFN — ang. *Page Frame Number*)).

Uwaga. Ze względów wydajnościowych hipernadzorca, budując mapowanie pamięci fizycznej, jest w stanie wykryć duże kawałki przyległej pamięci fizycznej i, w podobny sposób jak w przypadku pamięci wirtualnej, jest w stanie mapować te kawałki przy użyciu dużych stron. Jeśli z jakiegoś powodu system operacyjny uruchomiony w partycji zdecyduje się zastosować bardziej granularną ochronę strony fizycznej, hipernadzorca użyłby zarezerwowanej pamięci do złamania dużej strony w tablicy SLAT.

Wcześniejsze wersje hipernadzorcy obsługiwały również inną technikę mapowania fizycznej przestrzeni adresowej partycji: pseudo-stronicowanie(ang. *shadow paging*). Pseudo-stronicowanie było używane w przypadku tych maszyn, które nie posiadały obsługi SLAT. Technika ta miała bardzo duży negatywny wpływ na wydajność; w rezultacie nie jest już obsługiwana. (Maszyna musi obsługiwać SLAT; w przeciwnym razie hipernadzorca odmówiłby uruchomienia).

Tablica SLAT partycji głównej jest budowana w czasie tworzenia partycji, ale w przypadku partycji gościa sytuacja jest nieco inna. Gdy tworzona jest partycja podrzędna, hipernadzorca tworzy jej początkową fizyczną przestrzeń adresową, ale przydziela tylko tablicę stron partycji głównej (PML4) dla VTL-a każdej partycji. Przed uruchomieniem nowej maszyny wirtualnej

sterownik VID (część stosu wirtualizacji) rezerwuje strony fizyczne potrzebne dla maszyny wirtualnej (dokładna liczba zależy od rozmiaru pamięci maszyny wirtualnej), przydzielając je z partycji głównej. (Pamiętaj, że mówimy o pamięci fizycznej; tylko sterownik może przydzielać strony fizyczne). Sterownik VID utrzymuje listę stron fizycznych, która jest analizowana i dzielona na duże strony, a następnie jest wysyłana do hipernadzorcy poprzez hiperwywołanie powtarzalne *HvMapGpaPages*.

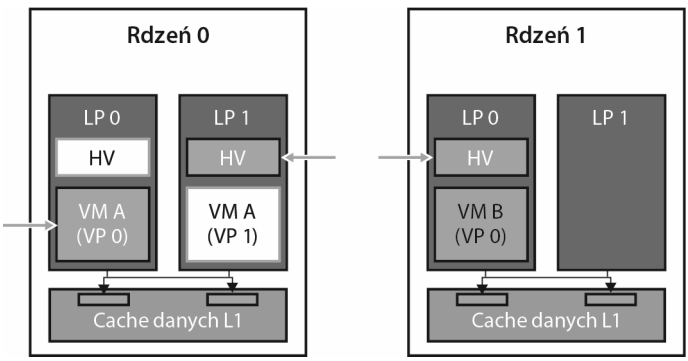
Przed wysłaniem żądania mapowania sterownik VID wysyła do hipernadzorcy wywołanie powodujące tworzenie potrzebnych tablic stron SLAT i wewnętrznych struktur danych przestrzeni pamięci fizycznej. Każda hierarchia tablic stron SLAT jest przydzielana dla każdego dostępnego VTL-a w partycji (nazywane jest to wstępnym zadeklarowaniem (ang. *pre-commit*)). Operacja ta może zakończyć się niepowodzeniem, np. gdy przedział nowej partycji nie mógł zawierać wystarczającej liczby stron fizycznych. W takim przypadku, jak omówiono w poprzednim punkcie, sterownik VID przydziela więcej pamięci z partycji głównej i deponuje ją w przedziale partycji podrzędnej. Na tym etapie sterownik VID może swobodnie mapować wszystkie strony fizyczne partycji podrzędnej. Hipernadzorca buduje i kompiluje wszystkie potrzebne tablice stron SLAT, przypisując im różne zabezpieczenia w zależności od poziomu VTL-a. (Duże strony wymagają o jeden poziom mniej pośredniczenia). Ten krok kończy tworzenie fizycznej przestrzeni adresowej partycji podrzędnej.

Izolacja przestrzeni adresowej

Odkryte w nowoczesnych procesorach podatności na wykonywanie spekulatywne (znane również jako *Meltdown*, *Spectre* i *Foreshadow*) pozwalały atakującemu na odczytanie tajnych danych znajdujących się w bardziej uprzywilejowanym kontekście wykonawczym poprzez spekulatywny odczyt nieaktualnych danych znajdujących się w pamięci podręcznej procesora. Oznacza to, że oprogramowanie wykonywane w maszynie wirtualnej gościa mogło potencjalnie być w stanie spekulatywnie odczytać prywatną pamięć należącą do hipernadzorcy lub bardziej uprzywilejowanej partycji głównej. Wewnętrzne szczegóły dotyczące *Spectre*, *Meltdown* i wszystkich podatności typu *side-channel* oraz sposób łagodzenia skutków ich działania przez Windows zostały szczegółowo omówione w rozdziale 8.

Hipernadzorca zdołał złagodzić skutki większości tego typu ataków dzięki zaimplementowaniu złagodzenia typu **HyperClear**. Złagodzenie typu HyperClear opiera się na trzech kluczowych komponentach zapewniających silną izolację między maszynami wirtualnymi, takich jak: planista rdzenia, izolacja przestrzeni adresowej procesora wirtualnego oraz oczyszczanie wrażliwych danych. W nowoczesnych procesorach wielordzeniowych często różne wątki wielowątkowości symetrycznej (SMT — ang. *Symmetric Multithreading*) korzystają z tej samej pamięci podręcznej procesora. (Szczegóły dotyczące planisty rdzenia i wielowątkowości symetrycznej znajdują się w punkcie „Planiści platformy Hyper-V”). W środowisku wirtualizacji wątki SMT działające na rdzeniu mogą niezależnie wchodzić i wychodzić z kontekstu hipernadzorcy na podstawie swojej aktywności. Na przykład zdarzenia takie jak przerwania mogą spowodować, że wątek SMT przestanie wykonywać kontekst wirtualnego procesora gościa i zacznie wykonywać kontekst hipernadzorcy. Może się to zdarzyć niezależnie dla każdego wątku SMT, więc jeden wątek SMT może być wykonywany w kontekście hipernadzorcy, podczas gdy jego wątek SMT nadal działa w kontekście wirtualnego procesora gościa maszyny wirtualnej. Atakujący, który uruchamia kod w mniej zaufanym kontekście wirtualnego procesora gościa maszyny wirtualnej na jednym z wątków SMT, może wykorzystać lukę w kanale bocznym, aby potencjalnie zaobserwować wrażliwe dane z kontekstu hipernadzorcy uruchomionego na równoległym wątku SMT.

Hipernadzorca zapewnia silną izolację danych w celu ochrony przed złośliwą maszyną wirtualną gościa poprzez utrzymywanie oddzielnych zakresów adresów wirtualnych dla każdego wątku SMT gościa (który obsługuje wirtualny procesor). Kiedy kontekst hipernadzorcy jest wprowadzany na konkretny wątek SMT, żadne tajne dane nie są adresowalne. Jedyne dane, które można wprowadzić do pamięci podręcznej CPU, są związane z tym bieżącym procesorem wirtualnym gościa lub odzwierciedlają współdzielone dane hipernadzorcy. Jak pokazano na rysunku 9.9, kiedy procesor wirtualny uruchomiony na wątku SMT wchodzi do hipernadzorcy, wymuszane jest (przez planistę głównej partycji), żeby równoległy procesor logiczny uruchamiał inny procesor wirtualny należący do tej samej maszyny wirtualnej. Ponadto w hipernadzorcy nie są mapowane żadne współdzielone tajne informacje. W przypadku gdy hipernadzorca potrzebuje dostępu do tajnych danych, zapewnia, że żaden inny procesor wirtualny nie jest zaplanowany w innym równoległym wątku SMT.

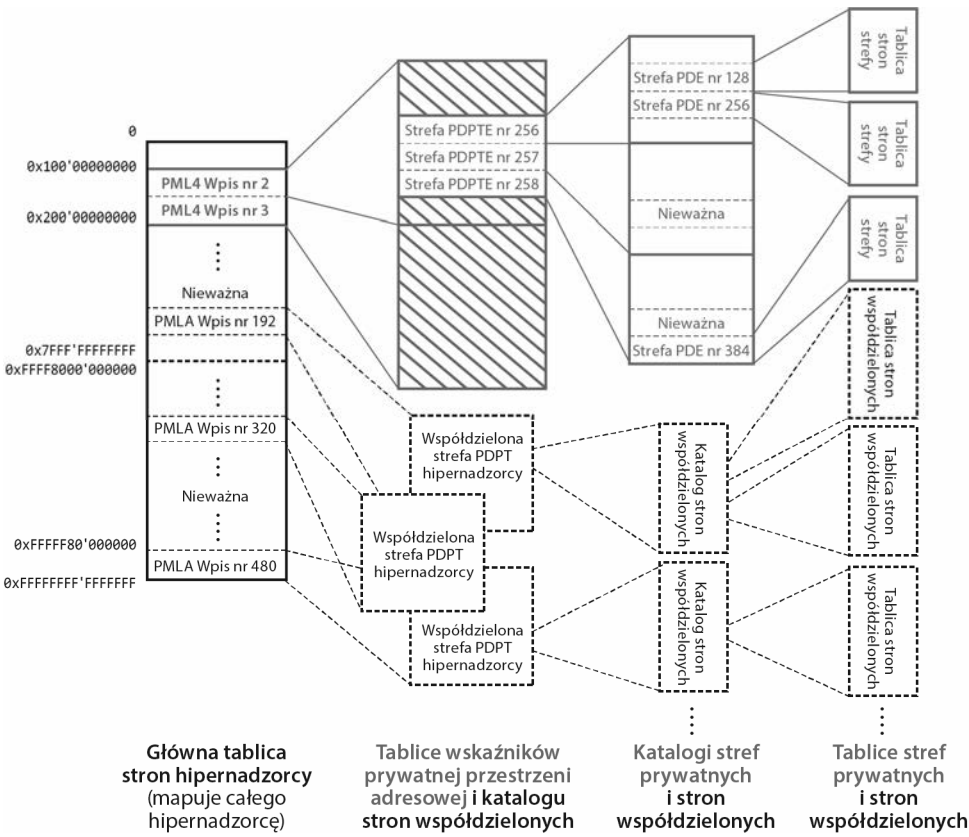


RYSUNEK 9.9. *Łagodzenie skutków typu Hyperclear*

W przeciwieństwie do jądra NT, hipernadzorca zawsze działa z partycją główną (*root*) posiadającą jednostronicową tablicę, co tworzy pojedynczą globalną wirtualną przestrzeń adresową. Hipernadzorca definiuje pojęcie prywatnej przestrzeni adresowej, która ma mylącą nazwę. W rzeczywistości hipernadzorca rezerwuje dwa globalne wpisy tablicy stron partycji głównej (*root*) (wpisy PML4, które generują zakres adresów wirtualnych 1 TB) dla mapowania lub usuwania mapowania prywatnej przestrzeni adresowej. Kiedy hipernadzorca początkowo konstruuje procesor wirtualny, przydziela dwa prywatne wpisy w tablicy stron w partycji głównej (*root*). Będą one używane do mapowania tajnych danych procesora wirtualnego, takich jak jego stos i struktury danych, które zawierają prywatne dane. Przełączenie przestrzeni adresowej oznacza dokonanie dwóch globalnych wpisów w tablicy stron w partycji głównej (*root*) (co wyjaśnia, dlaczego termin *prywatna przestrzeń adresowa* ma mylącą nazwę — tak naprawdę jest to prywatny *zakres* adresów). Hipernadzorca przełącza prywatne przestrzenie adresowe tylko w dwóch przypadkach: gdy tworzony jest nowy procesor wirtualny oraz podczas przełączania wątków. (Pamiętaj, że wątki są obsługiwane przez procesor wirtualny. Planista rdzenia zapewnia, że żaden z równoległych wątków SMT nie wykonuje instrukcji procesorów wirtualnych z różnych partycji). Podczas wykonywania kodu, wątek hipernadzorcy zmapował tylko prywatne dane swojego procesora wirtualnego; żadne inne tajne dane nie są dostępne dla tego wątku.

Mapowanie tajnych danych w prywatnej przestrzeni adresowej odbywa się za pomocą strefy pamięci, odzwierciedlanej przez strukturę danych *MM_ZONE*. Strefa pamięci hermetyzuje prywatny podzakres adresów wirtualnych prywatnej przestrzeni adresowej, w którym hipernadzorca zwykle przechowuje tajne dane każdego procesora wirtualnego.

Strefa pamięci działa podobnie jak prywatna przestrzeń adresowa. Zamiast mapować wpisy w globalnej tablicy stron partycji głównej (*root*), strefa pamięci mapuje prywatne katalogi stron w dwóch wpisach w partycji głównej, używanych przez prywatną przestrzeń adresową. Strefa pamięci utrzymuje tablicę katalogów stron, które będą mapowane i niemapowane do prywatnej przestrzeni adresowej, oraz bitmapę, która śledzi używane tablice stron. Rysunek 9.10 pokazuje związek między prywatną przestrzenią adresową a strefą pamięci. Strefy pamięci mogą być mapowane i niemapowane na żądanie (w prywatnej przestrzeni adresowej), ale zwykle są przełączane tylko w czasie tworzenia procesora wirtualnego. W rzeczywistości hipernadzorca nie musi ich przełączać podczas przełączania wątków; prywatna przestrzeń adresowa hermetyzuje zakres adresów wirtualnych eksponowany przez strefę pamięci.



RYSUNEK 9.10. *Prywatne przestrzenie adresowe hipernadzorcy i prywatne strefy pamięci*

Na rysunku 9.10 struktury tablicy stron związane z prywatną przestrzenią adresową wypełnione są wzorem, te związane ze strefą pamięci pokazane są w kolorze szarym, a te współdzielone, należące do hipernadzorcy, narysowane są linią przerywaną. Przełączanie prywatnych przestrzeni adresowych jest stosunkowo tanią operacją, która wymaga modyfikacji dwóch wpisów PML4 w głównym elemencie tablicy stron hipernadzorcy. Dołączenie lub odłączenie strefy pamięci z prywatnej przestrzeni adresowej wymaga jedynie modyfikacji segmentu PDPTE strefy (rozmiar adresów wirtualnych VA strefy jest zmienny; segmenty PDTPE są zawsze przydzielane w sąsiadujących segmentach).

Pamięć dynamiczna

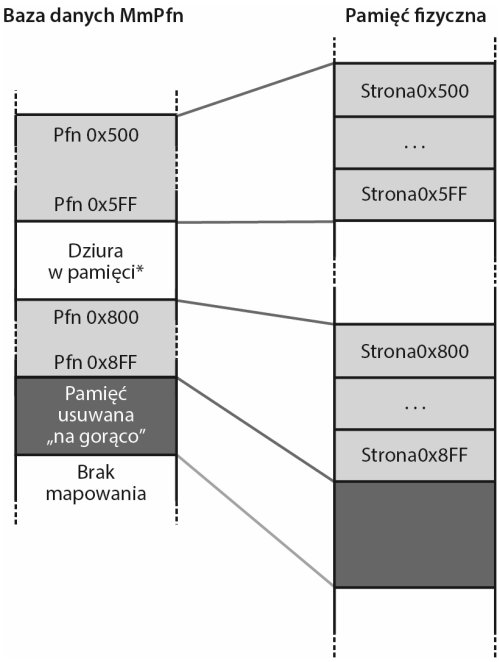
Maszyny wirtualne mogą wykorzystywać różny procent przydzielonej im pamięci fizycznej. Na przykład niektóre maszyny wirtualne używają tylko niewielkiej ilości przydzielonej im pamięci fizycznej gościa, zachowując dużą jej część w stanie uwolnionym lub wyzerowanym. Wydajność innych maszyn wirtualnych może natomiast ucierpieć w przypadku scenariuszy wysokiego obciążenia pamięci, gdzie plik stron jest używany zbyt często, ponieważ przydzielona pamięć fizyczna gościa nie jest wystarczająca. Aby zapobiec opisanemu scenariuszowi, hipernadzorca i stos wirtualizacji obsługują koncepcję (mechanizm) pamięci dynamicznej. *Pamięć dynamiczna* to możliwość dynamicznego przydzielania i usuwania pamięci fizycznej do i z maszyny wirtualnej. Funkcja ta jest zapewniana przez wiele komponentów, takich jak:

- Menedżer pamięci jądra NT, który obsługuje dodawanie i usuwanie pamięci fizycznej na gorąco (również w systemie bare-metalowym);
- Hipernadzorca, poprzez SLAT (zarządzany przez menedżera adresów);
- Proces roboczy maszyny wirtualnej, który używa modułu kontrolera pamięci dynamicznej — *Vmdynmem.dll* — dla nawiązania połączenia ze sterownikiem klienta usługi wirtualizacji pamięci dynamicznej VMBus (*Dmvsc.sys*), który działa na partycji podrzędnej.

Aby właściwie opisać pamięć dynamiczną, powinniśmy szybko przedstawić, w jaki sposób baza danych numerów ramek stron PFN jest tworzona przez jądro NT. Baza PFN jest używana przez Windows do śledzenia pamięci fizycznej. Została ona szczegółowo omówiona w rozdziale 5. części I. W celu utworzenia bazy PFN jądro NT najpierw oblicza hipotetyczny rozmiar potrzebny do odwzorowania najwyższego możliwego adresu fizycznego (256 TB w standardowych systemach 64-bitowych), a następnie oznacza przestrzeń adresów wirtualnych VA potrzebną do jej całkowitego odwzorowania jako zarezerwowaną (zapisując adres bazowy do zmiennej globalnej *MmPfnDatabase*). Zauważ, że zarezerwowana przestrzeń VA nadal nie ma przydzielonych tablic stron. Jądro NT cyklicznie przechodzi (używając usług UEFI) pomiędzy każdym deskryptorem pamięci fizycznej odkrytym przez menedżera rozruchu, łączy je w najdłuższe możliwe zakresy i dla każdego zakresu mapuje bazowe wpisy bazy PFN, używając dużych stron. Powoduje to ważne następstwa; jak widać na rysunku 9.11, baza danych PFN ma miejsce na największą możliwą ilość pamięci fizycznej, ale tylko niewielki jej podzbiór jest mapowany do rzeczywistych stron fizycznych (ta technika jest nazywana *pamięcią rzadką*).

Dodawanie i usuwanie pamięci fizycznej „na gorąco” działa dzięki tej zasadzie. Gdy do systemu dodawana jest nowa pamięć fizyczna, sterownik pamięci Plug and Play (Pnpmem.sys) wykrywa ją i wywołuje program *MmAddPhysicalMemory*, który jest eksportowany przez jądro NT. Jądro NT uruchamia złożoną procedurę, która oblicza dokładną liczbę stron w nowym zakresie i węzeł Numa, do którego należą, a następnie odwzorowuje nowe wpisy PFN w bazie danych, tworząc niezbędne tablice stron w zarezerwowanej przestrzeni VA. Nowe strony fizyczne są dodawane do listy wolnych (więcej szczegółów w rozdziale 5. w części I).

Gdy część pamięci fizycznej jest usuwana na gorąco, system wykonuje odwrotną procedurę. Sprawdza, czy strony należą do właściwej listy stron fizycznych, uaktualnia wewnętrzne liczniki pamięci (takie jak całkowita liczba stron fizycznych), a na koniec zwalnia odpowiednie wpisy PFN, co oznacza, że wszystkie zostaną oznaczone jako „złe”. Menedżer pamięci nie będzie już nigdy używał opisanych przez nie stron fizycznych. Z bazy danych PFN nie jest odmapowywana żadna faktyczna przestrzeń wirtualna. Pamięć fizyczna, która była opisana przez uwolnione PFN-y, zawsze może zostać ponownie dodana w przyszłości.



RYSUNEK 9.11. *Przykład bazy danych PFN, w której usunięto część pamięci fizycznej*

Podczas uruchamiania „oświeconej” maszyny wirtualnej sterownik pamięci dynamicznej (*Dmvsc.sys*) wykrywa, czy macierzysta maszyna wirtualna obsługuje funkcję dodawania na gorąco (ang. *hot add*); jeśli tak, to tworzy wątek roboczy, który negocjuje protokół i łączy się z kanałem VMBus dostawcy usługi wirtualizacji (VSP — ang. *Virtualization Service Provider*). (Szczegóły dotyczące klienta usługi wirtualizacji VSC (ang. *Virtualization Service Client*) i VSP znajdują się w podrozdziale „Stos wirtualizacji” w dalszej części tego rozdziału). Kanał połączenia VMBus łączy sterownik pamięci dynamicznej uruchomiony w partycji podrzędnej z modułem kontrolera pamięci dynamicznej (*Vmdynmem.dll*), który jest odwzorowany w procesie roboczym maszyny wirtualnej w partycji głównej. Uruchomiony zostaje protokół wymiany komunikatów. Co jedną sekundę partycja podrzędna pobiera raport o obciążeniu pamięci poprzez odpytywanie różnych liczników wydajnościowych udostępnianych przez menedżera pamięci (globalne wykorzystanie pliku strony, liczba dostępnych, zadeklarowanych i brudnych stron, liczba błędów strony na sekundę, liczba stron na liście stron wolnych i wyzerowanych). Raport ten jest następnie wysyłany do partycji głównej.

Proces roboczy maszyny wirtualnej w partycji głównej wykorzystuje usługi wyeksponowane przez balanser usługi zarządzania maszynami wirtualnymi (VMMS — ang. *Virtual Machine Management Service*), składnik usługi VmCompute, do obliczeń niezbędnych do określenia możliwości wykonania operacji dodawania na gorąco. Jeśli stan pamięci partycji głównej pozwolił na wykonanie operacji dodawania na gorąco, balanser usługi VMMS oblicza odpowiednią liczbę stron do zdeponowania w partycji podrzędnej i tworzy wywołanie zwrotne (poprzez COM) do procesu roboczego maszyny wirtualnej, który rozpoczyna operację dodawania na gorąco za pomocą sterownika VID:

1. Rezerwuje odpowiednią ilość pamięci fizycznej w partycji głównej.
2. Wywołuje hipernadzorcę w celu zmapowania systemowych stron fizycznych zarezerwowanych przez partycję główną do niektórych stron fizycznych gościa zmapowanych w podrzędnej maszynie wirtualnej, z odpowiednim zabezpieczeniem.
3. Wysyła wiadomość do sterownika pamięci dynamicznej w celu rozpoczęcia operacji dodawania „na gorąco" na niektórych stronach fizycznych gościa, zmapowanych wcześniej przez hipernadzorcę.

Sterownik pamięci dynamicznej w partycji potomnej używa API *MmAddPhysicalMemory* eksponowanego przez jądro NT do wykonania operacji dodawania „na gorąco". Ta ostatnia operacja mapuje PFN-y opisujące nową pamięć fizyczną gościa w bazie PFN, dodając w razie potrzeby nowe strony wspierające do bazy.

W podobny sposób, gdy balanser VMMS wykryje, że podrzędna maszyna wirtualna ma dużo dostępnych stron fizycznych, może zażądać od partycji podrzędnej (jeszcze poprzez proces roboczy maszyny wirtualnej) usunięcia na gorąco niektórych stron fizycznych. Sterownik pamięci dynamicznej wykorzystuje API *MmRemovePhysicalMemory* do wykonania operacji usuwania na gorąco (ang. *hot remove*). Jądro NT sprawdza, czy każda strona w zakresie określonym przez balanser znajduje się na liście wyzerowanej lub wolnej albo należy do stosu, który może być bezpiecznie stronicowany. Jeśli wszystkie warunki zostaną spełnione, sterownik pamięci dynamicznej odsyła zakres stron do usunięcia „na gorąco" do procesu roboczego maszyny wirtualnej, który za pomocą usług dostarczanych przez sterownik VID usunie mapowanie stron fizycznych z partycji podrzędnej i uwolni je z powrotem do jądra NT.

Uwaga. Pamięć dynamiczna nie jest obsługiwana, gdy włączona jest wirtualizacja zagnieżdżona.

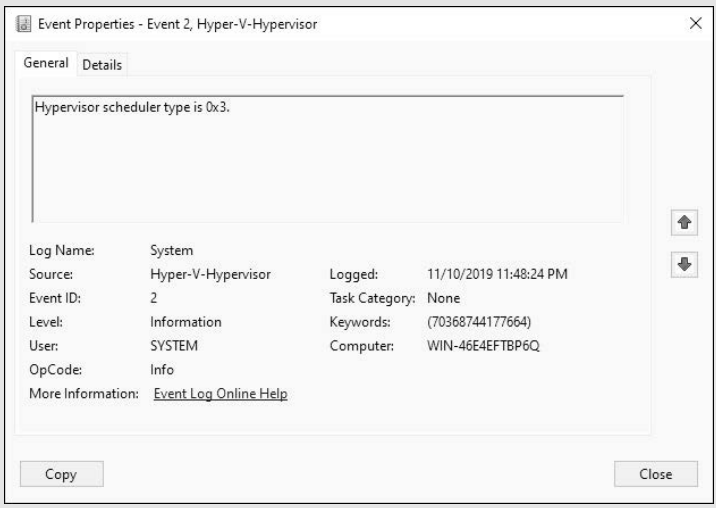
Planiści platformy Hyper-V

Hipernadzorca jest rodzajem mikrosystemu operacyjnego, który działa poniżej systemu operacyjnego partycji głównej (Windows). Jako taki powinien być w stanie zdecydować, który wątek (wspierający procesor wirtualny) jest wykonywany przez który procesor fizyczny. Jest to szczególnie istotne, gdy w systemie działa wiele maszyn wirtualnych złożonych w sumie z większej liczby procesorów wirtualnych niż procesorów fizycznych zainstalowanych w stacji roboczej. Rolą planisty hipernadzorcy jest wybór kolejnego wątku, który fizyczny procesor wykonuje, skoro tylko wycinek czasu przydzielony dla bieżącego wątku się zakończy. Platforma Hyper-V może używać trzech różnych planistów. Aby odpowiednio zarządzać wszystkimi różnymi planistami, hipernadzorca eksponuje *API planisty* — zestaw programów, które są jedynymi punktami wejścia do planisty hipernadzorcy. Ich wyłącznym celem jest przekierowanie wywołań API do konkretnej implementacji planisty.

Eksperyment: kontrolowanie typu planisty hipernadzorcy

Podczas gdy edycje klienckie systemu Windows uruchamiają się domyślnie z planistą głównym (*root*), system Windows Server 2019 działa domyślnie z planistą rdzenia. W tym eksperymencie zorientujesz się, jaki planista hipernadzorcy jest włączany w Twoim systemie, i dowiesz się, jak przełączyć się na inny rodzaj planisty hipernadzorcy przy następnym restarcie systemu.

Hipernadzorca Windows rejestruje zdarzenie systemowe po określeniu, którego z planistów należy włączyć. Zarejestrowane zdarzenie można przeszukać za pomocą narzędzia o nazwie *Event Viewer* (*Podgląd zdarzeń*), które można uruchomić, wpisując `eventvwr` w polu wyszukiwania na pasku zadań. Po uruchomieniu apletu rozwiń klucz *Windows Logs* (*Dzienniki systemu Windows*) i kliknij *System log* (*System*). Wyszukaj zdarzenia o ID 2 i źródłach zdarzeń ustawionych na Hyper-V-Hypervisor. Możesz to zrobić, klikając przycisk *Filter Current Log* (*Filtruj bieżący dziennik*) znajdujący się po prawej stronie okna lub klikając kolumnę *Event ID* (*Identyfikator zdarzenia*), która uporządkuje zdarzenia w kolejności rosnącej według ich Identyfikatora (pamiętaj, że operacja może chwilę potrwać). Jeśli dwukrotnie klikniesz znalezione zdarzenie, powinieneś zobaczyć okno takie jak poniżej:



ID zdarzenia startowego 2 oznacza w rzeczywistości typ planisty hipernadzorcy, gdzie:

1 = Planista klasyczny, SMT wyłączony,
2 = Planista klasyczny,
3 = Planista rdzenia,
4 = Planista główny (*root*).

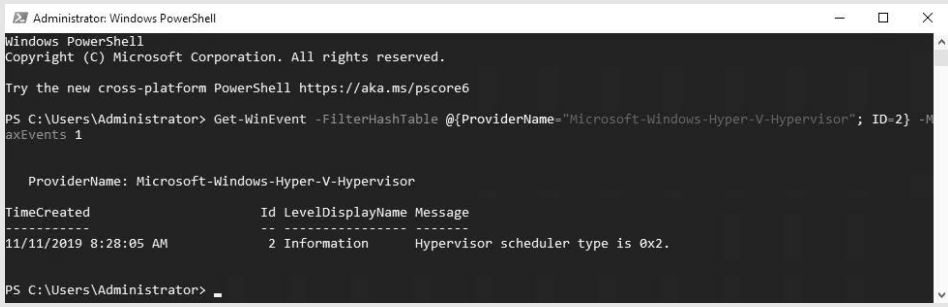
Przykładowy rysunek pochodzi z systemu Windows Server, który domyślnie pracuje z planistą rdzenia (ang. *Core Scheduler*). Aby zmienić typ planisty na klasyczny (lub główny (*root*)), otwórz administracyjne okno wiersza poleceń (wpisz **`cmd`** w polu wyszukiwania na pasku zadań, następnie kliknij prawym przyciskiem myszy ikonę *Command Prompt* (*Wiersz polecenia*) i wybierz opcję *Run as administrator* (*Uruchom jako administrator*)) i wpisz następujące polecenie:

```
bcdedit /set hypervisorschedulertype <typ>
```

gdzie typ <*typ*> to `Classic` w przypadku klasycznego planisty, `Core` w przypadku planisty rdzenia lub `Root` w przypadku planisty głównego (*root*). Uruchom ponownie system i sprawdź nowo wygenerowany identyfikator zdarzenia Hyper-V-Hypervisor Id 2. Można również sprawdzić aktualnie włączonego planistę hipernadzorcy za pomocą administracyjnego okna PowerShell, wpisując następujące polecenie:

```
Get-WinEvent -FilterHashTable @{ProviderName="Microsoft-Windows-Hyper-V-Hypervisor";
↪ID=2} -MaxEvents 1
```

Polecenie wyodrębnia z dziennika zdarzeń systemu ostatnie zdarzenie o ID 2.

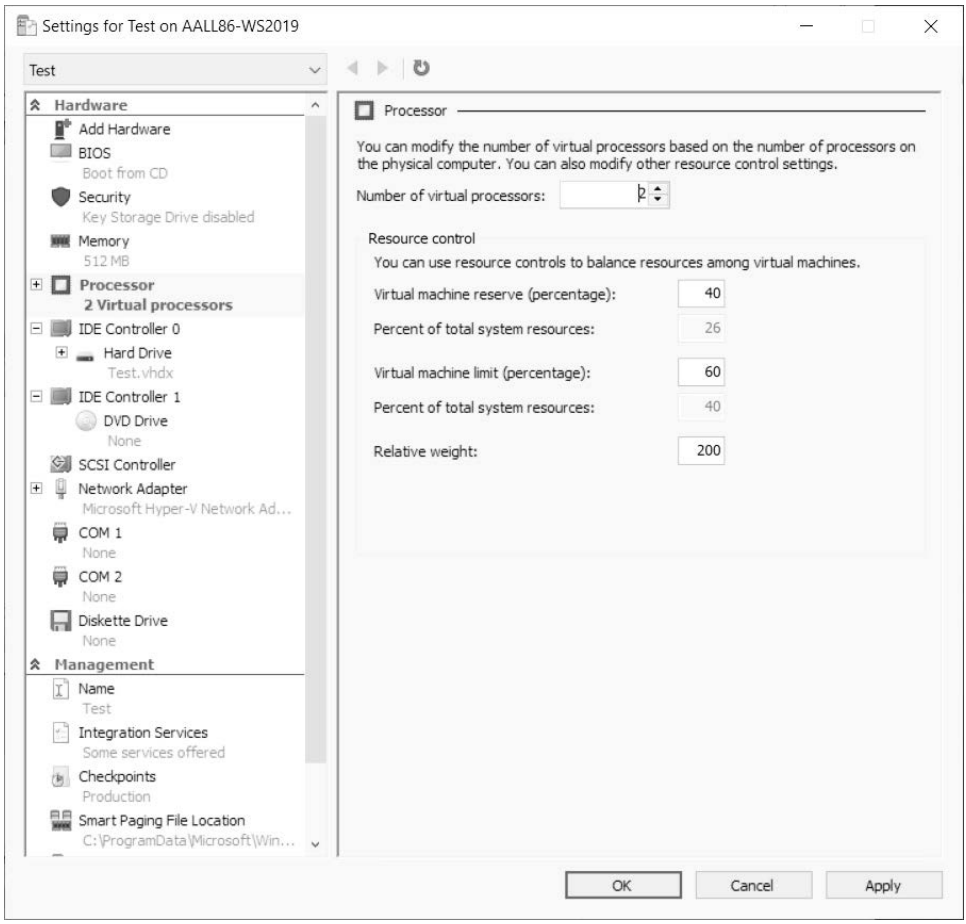


Planista klasyczny

Planista klasyczny jest domyślnym planistą używanym we wszystkich wersjach platformy Hyper-V od czasu jego pierwszego wydania. Planista klasyczny w swojej domyślnej konfiguracji implementuje prostą politykę tzw. *round-robin*, w której każdy wirtualny procesor w aktualnym stanie wykonania (stan wykonania zależy od całkowitej liczby maszyn wirtualnych uruchomionych w systemie) ma równe szanse na to, aby jego instrukcje zostały rozesłane. Klasyczny planista obsługuje również ustawianie powinowactwa wirtualnego procesora i wykonuje decyzje dotyczące planowania, biorąc pod uwagę węzeł NUMA procesora fizycznego. Klasyczny planista nie wie, co aktualnie wykonuje procesor wirtualny gościa. Jedyny wyjątek jest zdefiniowany przez mechanizm rozszerzenia „oświecającego" typu *spin-lock*. Gdy jądro systemu Windows, działające w partycji, ma zamiar wykonać aktywne oczekiwanie na spin-lock, emituje hiperwywołanie, mając na celu poinformowanie o tym hipernadzorcy (mechanizmy synchronizacji wysokiego IRQL zostały opisane w rozdziale 8., „Mechanizmy systemowe"). Klasyczny planista może uprzedzić procesor wirtualny aktualnie wykonujący instrukcje (któremu nie upłynął jeszcze przydzielony wycinek czasu) i zaplanować kolejny. W ten sposób oszczędza aktywne cykle spinowe procesora.

Domyślna konfiguracja klasycznego planisty przypisuje każdemu wirtualnemu procesorowi równy wycinek czasu. Oznacza to, że w nadsubskrybowanych systemach o dużym obciążeniu, gdzie wiele wirtualnych procesorów próbuje wykonać operacje, a fizyczne procesory są wystarczająco zajęte, wydajność może szybko spaść. Aby przezwyciężyć ten problem, klasyczny planista obsługuje różne opcje dostrajania (patrz rysunek 9.12), które mogą modyfikować jego wewnętrzne decyzje dotyczące planowania:

- **Rezerwacje procesora wirtualnego** — Użytkownik może z góry zarezerwować pojemność procesora w imieniu maszyny gościa. Rezerwacja jest określana jako procent pojemności procesora fizycznego, który ma być udostępniony maszynie gościa zawsze, gdy zostanie zaplanowane jej uruchomienie. W rezultacie platforma Hyper-V planuje uruchomienie procesora wirtualnego tylko wtedy, gdy ta minimalna ilość pojemności procesora jest dostępna (co oznacza, że przydzielony wycinek czasu jest gwarantowany).
- **Limity procesora wirtualnego** — Podobnie jak w przypadku rezerwacji procesorów wirtualnych użytkownik może ograniczyć procent wykorzystania fizycznego procesora dla procesora wirtualnego. Oznacza to zmniejszenie dostępnego wycinka czasu przydzielonego dla procesora wirtualnego w scenariuszu dużego obciążenia.



RYSUNEK 9.12. *Strona właściwości ustawień planisty klasycznego, która jest dostępna tylko wtedy, gdy włączony jest planista klasyczny*

- **Waga procesora wirtualnego** — To ustawienie kontroluje prawdopodobieństwo, że procesor wirtualny jest zaplanowany, gdy rezerwacje zostały już zrealizowane. W domyślnych konfiguracjach każdy procesor wirtualny ma równe prawdopodobieństwo, że jego instrukcje będą wykonywane. Kiedy użytkownik skonfiguruje wagę na procesorze wirtualnym, które należą do maszyny wirtualnej, decyzje dotyczące planowania zaczynają być opierane na odpowiednim współczynniku wagi, który wybrał użytkownik. Na przykład załóżmy, że w systemie z czterema procesorami działają jednocześnie trzy maszyny wirtualne. Pierwsza maszyna wirtualna ma ustawiony współczynnik wagowy 100, druga 200, a trzecia 300. Zakładając, że wszystkie fizyczne procesory systemu są przydzielone do jednolitej liczby procesorów wirtualnych, prawdopodobieństwo rozesłania do procesora wirtualnego w pierwszej maszynie wirtualnej wynosi 17%, procesora wirtualnego w drugiej maszynie wirtualnej — 33%, a procesora wirtualnego w trzeciej — 50%.

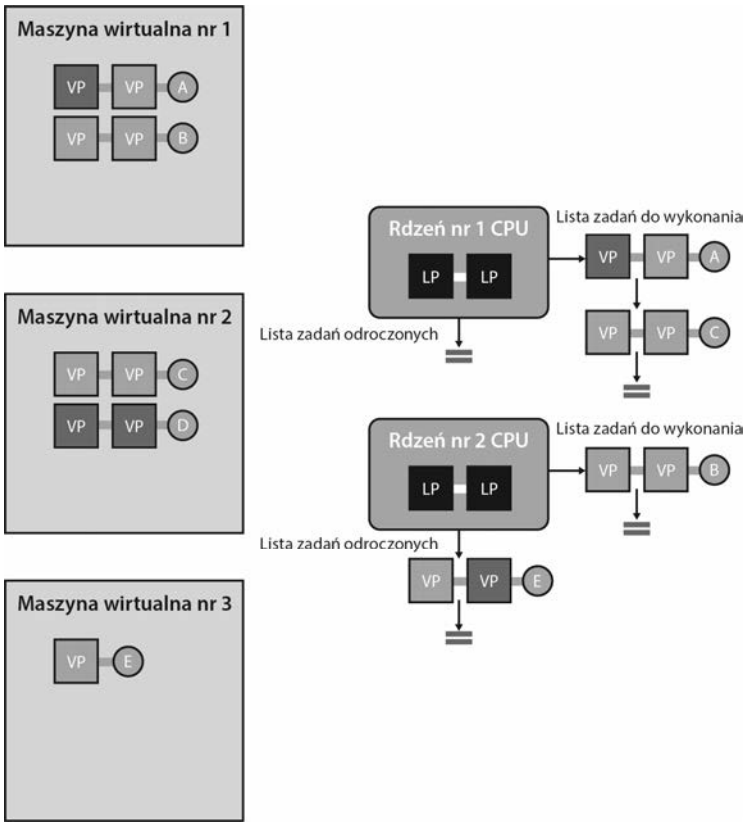
Planista rdzenia

W normalnych warunkach rdzeń klasycznego procesora posiada pojedynczy potok wykonawczy, w którym strumienie instrukcji są wykonywane jeden po drugim. Instrukcja wchodzi do potoku, przechodzi przez kilka etapów wykonania (np. załadowanie danych, obliczenie, zapisanie danych) i jest z niego usuwana. Różne typy instrukcji wykorzystują różne części rdzenia procesora. Rdzeń współczesnego procesora często jest w stanie wykonać poza kolejnością (ang. *out-of-order*) wiele instrukcji sekwencyjnych w potoku (w odniesieniu do kolejności, w jakiej weszły do potoku). Nowoczesne procesory, które obsługują wykonywanie poza kolejnością, często implementują coś, co nazywa się symetryczną wielowątkowością (SMT — ang. *Symmetric Multithreading*): rdzeń procesora ma dwa potoki wykonawcze i prezentuje systemowi więcej niż jeden procesor logiczny; w ten sposób dwa różne strumienie instrukcji mogą być wykonywane obok siebie przez jeden wspólny mechanizm wykonawczy (zasoby rdzenia, takie jak jego cache, są współdzielone). Dwa potoki wykonawcze są prezentowane oprogramowaniu (software) jako pojedyncze, niezależne procesory (CPU). Od tej pory, używając terminu *procesor logiczny* (lub po prostu LP — ang. *Logical Processor*), będziemy odnosić się do potoku wykonawczego rdzenia SMT prezentowanego systemowi Windows jako niezależny procesor. (SMT jest omówiona w rozdziałach 2. i 4. części I).

Taka implementacja sprzętowa doprowadziła do wielu problemów z bezpieczeństwem: jedna instrukcja wykonywana przez współdzielony logiczny CPU może zakłócić i wpłynąć na instrukcję wykonywaną przez inny równoległy procesor logiczny. Ponadto pamięć cache fizycznego rdzenia jest współdzielona; procesor logiczny może zmienić zawartość pamięci cache. Drugi z procesorów równoległych może potencjalnie sondować dane znajdujące się w pamięci podręcznej, mierząc czas potrzebny procesorowi na uzyskanie dostępu do pamięci adresowanej przez tę samą linię pamięci podręcznej, ujawniając w ten sposób „tajne dane", do których dostęp uzyskał drugi procesor logiczny (jak opisano w podrozdziale „Sprzętowe błędy w bocznym kanale" w rozdziale 8.). Klasyczny planista może normalnie wybrać dwa wątki należące do różnych maszyn wirtualnych do wykonania przez dwa procesory logiczne w tym samym rdzeniu procesora. Jest to oczywiście niedopuszczalne, ponieważ w tym kontekście pierwsza maszyna wirtualna mogłaby potencjalnie czytać dane należące do drugiej.

Aby pokonać ten problem i móc uruchamiać maszyny wirtualne z obsługą SMT z przewidywalną wydajnością, w systemie Windows Server 2016 wprowadzono planistę rdzenia. Planista rdzenia wykorzystuje właściwości SMT, aby zapewnić izolację i silną granicę bezpieczeństwa dla procesorów wirtualnych gości. Gdy planista rdzenia jest włączony, platforma Hyper-V rozplanowuje zadania rdzeni wirtualnych na rdzenie fizyczne. Ponadto zapewnia, że procesory wirtualne należące do różnych maszyn wirtualnych nigdy nie są zaplanowane na równoległych wątkach SMT rdzenia fizycznego. Planista rdzenia umożliwia maszynie wirtualnej wykorzystanie SMT. Procesory wirtualne prezentowane maszynie wirtualnej mogą być częścią zestawu SMT. System operacyjny i aplikacje działające w maszynie wirtualnej gościa mogą wykorzystywać zachowanie SMT i interfejsy programistyczne (API) do kontrolowania i rozdzielania pracy pomiędzy wątki SMT, tak jak w przypadku uruchomienia niezwirtualizowanego.

Rysunek 9.13 pokazuje przykład systemu SMT z czterema logicznymi procesorami rozmieszczonymi w dwóch rdzeniach CPU. Na rysunku uruchomione są trzy maszyny wirtualne. Pierwsza i druga maszyna wirtualna mają cztery procesory wirtualne w dwóch grupach po dwa, natomiast trzecia ma przypisany tylko jeden procesor wirtualny. Grupy procesorów wirtualnych w maszynach wirtualnych są oznaczone od A do E. Poszczególne procesory wirtualne w grupie, które są bezczynne (nie mają kodu do wykonania) są wypełnione ciemniejszym kolorem.

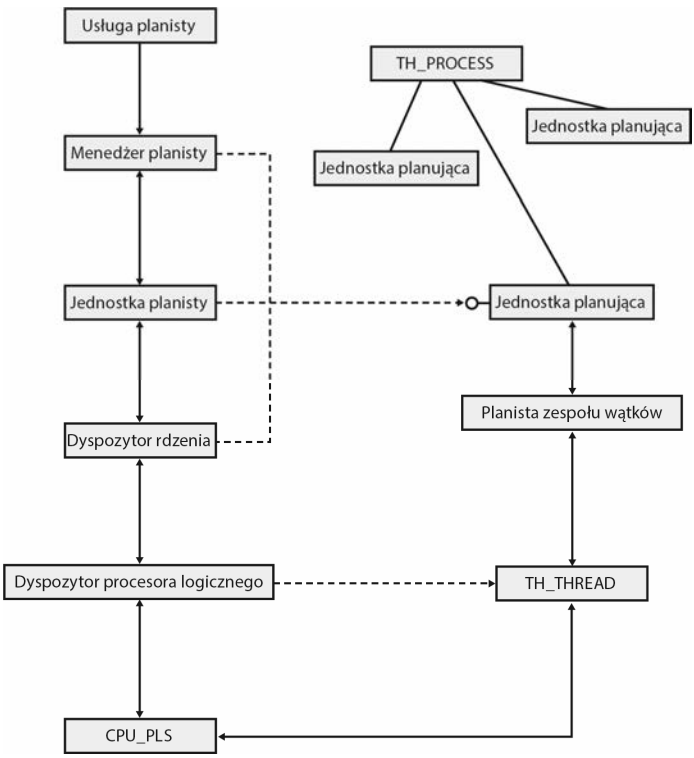


RYSUNEK 9.13. *Przykładowy system SMT z dwoma rdzeniami procesora i trzema uruchomionymi maszynami wirtualnymi*

Każdy rdzeń posiada listę wykonawczą zawierającą grupy procesorów wirtualnych, których instrukcje są gotowe do wykonania, oraz listę zadań odroczonych zawierającą grupy procesorów wirtualnych, których instrukcje są gotowe do wykonania, ale nie zostały jeszcze dodane do listy wykonawczej danego rdzenia. Instrukcje grupy procesorów wirtualnych są wykonywane na fizycznych rdzeniach. Jeśli wszystkie procesory wirtualne w grupie staną się bezczynne, to grupa procesorów wirtualnych zostaje zdeklasowana i nie pojawia się na żadnej liście wykonawczej. (Na rysunku 9.13 jest przedstawiona taka sytuacja dla grupy D procesorów wirtualnych). Jedyny procesor wirtualny z grupy E niedawno opuścił stan bezczynności. Ten procesor wirtualny został przydzielony do rdzenia nr 2 procesora. Na rysunku pokazana jest atrapa równoległych procesorów logicznych rdzenia. Dzieje się tak dlatego, że procesor logiczny rdzenia 2 nigdy nie planuje zadań dla żadnego innego procesora wirtualnego, podczas gdy równoległy do niego procesor logiczny pracujący na tym samym rdzeniu wykonuje instrukcje procesora wirtualnego należącego do maszyny wirtualnej 3. W ten sam sposób na rdzeniu fizycznym nie są planowane zadania żadnych innych procesorów wirtualnych, jeśli jeden procesor wirtualny z grupy procesorów logicznych stał się bezczynny, a instrukcje drugiego nadal są wykonywane (tak jak na przykład w przypadku grupy A). Każdy rdzeń wykonuje instrukcje grupy procesorów wirtualnych, która znajduje się na początku jego listy wykonawczej. Jeśli nie ma do wykonania instrukcji żadnych grup procesorów wirtualnych, rdzeń staje się bezczynny i czeka na grupę procesorów wirtualnych,

która zostanie umieszczona na jego odroczonej liście wykonawczej. Gdy to nastąpi, rdzeń budzi się z bezczynności i opróżnia swoją odroczoną listę wykonawczą, umieszczając jej zawartość na swojej liście wykonawczej.

Planista rdzenia jest implementowany przez różne komponenty (patrz rysunek 9.14), które zapewniają ścisłą warstwowość między sobą. Sercem planisty jest *jednostka planująca*, która odzwierciedla wirtualny rdzeń lub grupę SMT procesorów wirtualnych. (W przypadku maszyn wirtualnych niebędących SMT, odzwierciedla ona pojedynczy procesor wirtualny). W zależności od typu maszyny wirtualnej, jednostka planująca ma przypisany jeden lub dwa wątki. Proces hipernadzorcy posiada listę jednostek planujących, które mają swoje wątki wspierające procesory wirtualne należące do danej maszyny wirtualnej. Jednostka planująca jest to pojedyncza jednostka, która planuje zasoby dla planisty rdzenia (maszyny wirtualnej). Do planisty rdzenia (maszyny wirtualnej) — podczas wykonywania kodu — stosuje się ustawienia planowania takie jak rezerwacja, waga i limit. Jednostka planująca pozostaje aktywna przez czas trwania wycinka czasu, może być blokowana i odblokowywana oraz może migrować pomiędzy różnymi fizycznymi rdzeniami procesora. Ważną koncepcją jest to, że jednostka planująca jest analogiczna do wątku w klasycznym planiście, ale nie posiada stosu lub kontekstu procesora wirtualnego, w którym może działać. Jest to jeden z wątków związanych z jednostką planującą, która działa na fizycznym rdzeniu procesora. *Planista gangu wątków* jest arbitrem dla każdej jednostki podlegającej planowaniu. To podmiot, który decyduje o tym, który wątek z aktywnej jednostki planującej zostanie uruchomiony przez który procesor logiczny fizycznego rdzenia procesora. Wymusza on powinowactwa wątków, stosuje zasady planowania wątków i aktualizuje powiązane liczniki dla każdego wątku.



RYSUNEK 9.14. *Elementy składowe planisty rdzenia procesora*

Z każdym procesorem logicznym rdzenia procesora fizycznego związane jest wystąpienie dyspozytora procesora logicznego. *Dyspozytor procesora logicznego* jest odpowiedzialny za przełączanie wątków, utrzymywanie czasomierzy i „spłukiwanie" zawartości VMCS (lub VMCB, w zależności od architektury) dla bieżącego wątku. Dyspozytory procesorów logicznych są własnością *dyspozytora rdzenia*, który odzwierciedla fizyczny pojedynczy rdzeń procesora i posiada dokładnie dwa procesory logiczne SMT. Dyspozytor rdzenia zarządza bieżącą (aktywną) jednostką planującą. Planista jednostki, który jest związany z własnym dyspozytorem rdzenia, decyduje, która jednostka planująca musi być uruchomiona w następnej kolejności na fizycznym rdzeniu procesora, do którego należy planista jednostki. Ostatnim ważnym komponentem planisty jest *menedżer planisty*, który posiada wszystkich planistów w systemie i ma globalny wgląd w ich stany. Zapewnia on usługi równoważenia obciążenia i idealnego przypisania rdzenia do planisty jednostki.

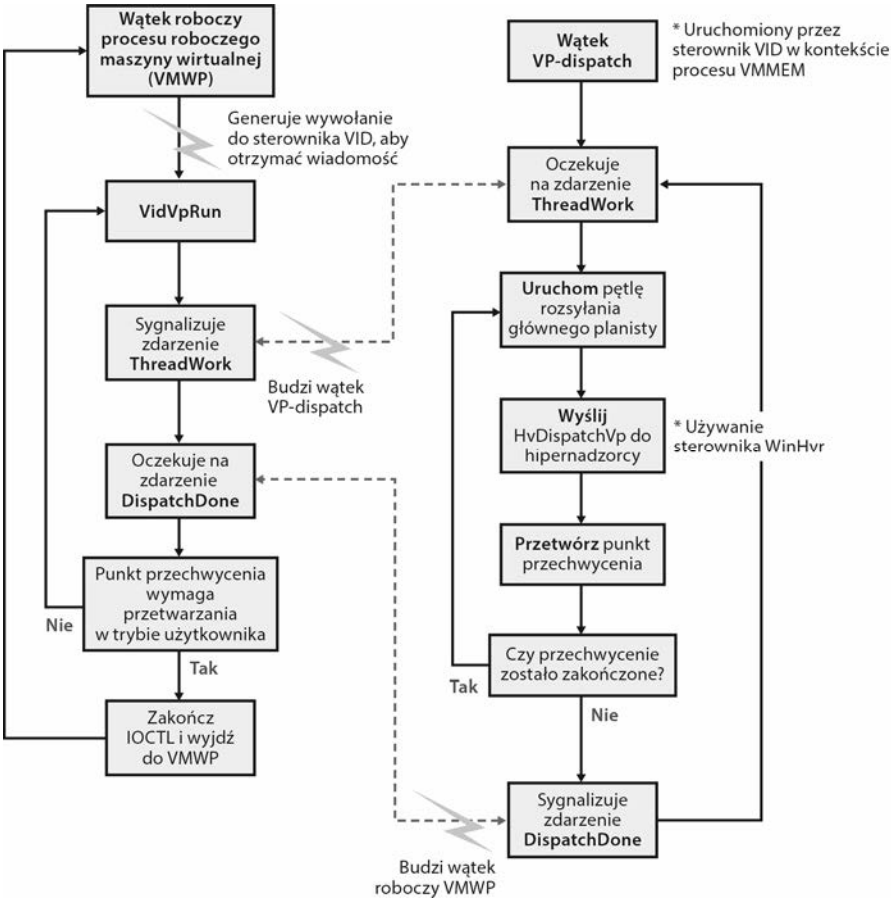
Główny planista (root scheduler)

Planista główny (znany również jako planista zintegrowany) został wprowadzony w aktualizacji systemu Windows 10 z kwietnia 2018 (RS4) w celu umożliwienia partycji głównej zaplanowania zadań dla procesorów wirtualnych należących do wielu partycji gości. Planista główny został zaprojektowany z myślą o obsłudze lekkich kontenerów używanych przez mechanizm Windows Defender Application Guard. Te typy kontenerów (wewnętrznie nazywane kontenerami Barcelona lub Krypton) muszą być zarządzane przez partycję główną i powinny zużywać niewielką ilość pamięci i miejsca na dysku twardym. (Dogłębne opisanie kontenerów Krypton wykracza poza zakres tej książki. Wprowadzenie do kontenerów serwerowych znajdziesz w części I, w rozdziale 3., „Procesy i zadania"). Ponadto główny (*root*) planista systemu operacyjnego może łatwo gromadzić metryki dotyczące wykorzystania procesora w kontenerze i używać tych danych jako danych wejściowych do tej samej polityki planowania, która ma zastosowanie do wszystkich innych obciążeń w systemie.

Planista NT w wystąpieniu systemu operacyjnego partycji głównej zarządza wszystkimi aspektami planowania pracy dla systemowych procesorów logicznych. Aby to osiągnąć, zintegrowany komponent planisty głównego w sterowniku VID tworzy wątek VP-dispatch, czyli wątek rozsyłania zadań do procesorów wirtualnych (ang. *VP-dispatch thread*) wewnątrz partycji głównej (w kontekście nowego procesu VMMEM) dla każdego procesora wirtualnego gościa. (Maszyny wirtualne obsługiwane przez Wirtualne adresy VA są omówione w dalszej części tego rozdziału). Planista NT w partycji głównej planuje wątki VP-dispatch jako zwykłe wątki podlegające dodatkowym zasadom planowania i rozszerzeniom „oświecającym" specyficznym dla maszyny wirtualnej / procesora wirtualnego. Każdy wątek rozsyłania zadań do procesorów wirtualnych uruchamia pętlę rozsyłania zadań do procesorów wirtualnych (ang. *VP-dispatch loop*), dopóki sterownik VID nie zakończy pracy odpowiedniego procesora wirtualnego.

Wątek VP-dispatch jest tworzony przez sterownik VID po tym, jak proces roboczy maszyny wirtualnej (VMWP — ang. *VM Worker Process*), o którym mowa w podrozdziale „Stos wirtualizacji", zażąda utworzenia partycji i procesorów wirtualnych poprzez *SETUP_PARTITION* IOCTL. Sterownik VID komunikuje się ze sterownikiem WinHvr, który z kolei inicjalizuje tworzenie partycji gościa przez hipernadzorcę (poprzez hiperwywołanie *HvCreatePartition*). W przypadku gdy utworzona partycja przedstawia maszynę wirtualną obsługiwaną przez wirtualne adresy VA lub gdy system ma aktywnego głównego planistę, sterownik VID wywołuje jądro NT (poprzez rozszerzenie jądra) w celu utworzenia minimalnego procesu VMMEM związanego z nową partycją gościa. Sterownik VID tworzy również wątek VP-dispatch dla każdego procesora wirtualnego należącego do partycji. Wątek VP-dispatch jest wykonywany w kontekście procesu

VMMEM w trybie jądra (w VMMEM nie istnieje kod trybu użytkownika) i jest zaimplementowany w sterowniku VID (oraz WinHvr). Jak pokazano na rysunku 9.15, każdy wątek VP-dispatch wykonuje pętlę VP-dispatch, dopóki VID nie zakończy pracy odpowiedniego procesora wirtualnego lub nie zostanie wygenerowane przechwycenie z partycji gościa.



RYSUNEK 9.15. *Wątek VP-dispatch głównego planisty i związany z nim wątek roboczy procesu roboczego maszyny wirtualnej (VMWP), który przetwarza komunikaty hipernadzorcy*

Podczas poruszania się po pętli VP-dispatch, wątek VP-dispatch jest odpowiedzialny za następujące czynności:

1. Wywołanie nowego interfejsu hiperwywołań hipernadzorcy *HvDispatchVp* w celu przydzielenia procesora wirtualnego do bieżącego procesora. Przy każdym hiperwywołaniu *HvDispatchVp* hipernadzorca próbuje przełączyć kontekst z bieżącego głównego procesora wirtualnego na określony procesor wirtualny gościa i pozwolić mu uruchomić kod gościa. Jedną z najważniejszych cech tego hiperwywołania jest to, że kod, który go emituje powinien działać na *PASSIVE_LEVEL* IRQL. Hipernadzorca pozwala procesorowi wirtualnemu gościa działać do momentu, gdy albo procesor wirtualny zablokuje się dobrowolnie, procesor wirtualny wygeneruje przechwycenie dla partycji głównej (*root*), albo nastąpi przerwanie skierowane do procesora wirtualnego partycji głównej (*root*).

Przerwania zegarowe są nadal przetwarzane przez partycje główne (*root*). Gdy procesor wirtualny gościa wyczerpie swój przydzielony wycinek czasu, wątek wspierający procesor wirtualny jest uprzedzany przez planistę NT. Przy którymkolwiek z tych trzech zdarzeń, hipernadzorca przełącza się z powrotem na główny (*root*) procesor wirtualny i kończy hiperwywołanie *HvDispatchVp*. Następnie wraca do partycji głównej.

2. Blokowanie na zdarzeniu VP-dispatch, jeśli odpowiadający mu procesor wirtualny w hipernadzorcy jest zablokowany. Za każdym razem, gdy procesor wirtualny gościa jest zablokowany dobrowolnie, wątek VP-dispatch blokuje się na zdarzeniu VP-dispatch, dopóki hipernadzorca nie odblokuje odpowiedniego procesora wirtualnego gościa i nie powiadomi sterownika VID. Sterownik VID sygnalizuje zdarzenie VP-dispatch, a planista NT odblokowuje wątek VP-dispatch, który może wykonać kolejne hiperwywołanie *HvDispatchVp*.
3. Przetwarzanie wszystkich przechwytów zgłoszonych przez hipernadzorcę po powrocie z hiperwywołania rozsyłania. Jeśli procesor wirtualny gościa wygeneruje przechwycenie dla partycji głównej (*root*), wątek VP-dispatch przetwarza żądanie przechwycenia po powrocie z hiperwywołania *HvDispatchVp* i wykonuje kolejne żądanie *HvDispatchVp* po zakończeniu przetwarzania przechwycenia przez VID. Każdy przechwyt jest zarządzany inaczej. Jeśli przechwyt wymaga przetworzenia przez Proces VMWP trybu użytkownika, sterownik WinHvr kończy pętlę i wraca do VID, który sygnalizuje zdarzenie dla obsługiwanego wątku VMWP i czeka na przetworzenie wiadomości przechwytującej przez proces VMWP przed ponownym uruchomieniem pętli.

Aby prawidłowo dostarczyć sygnały do wątków VP-dispatch z hipernadzorcy do partycji głównej (*root*), zintegrowany planista dostarcza mechanizm wymiany komunikatów z planistą. Hipernadzorca wysyła komunikaty planisty do partycji głównej za pośrednictwem strony współdzielonej. Kiedy nowa wiadomość jest gotowa do dostarczenia, hipernadzorca wstrzykuje do partycji głównej przerwanie SINT, a ta przekazuje je do odpowiedniego handlera ISR w sterowniku WinHvr, który kieruje wiadomość do wywołania zwrotnego przechwytu VID (*VidInterceptIsrCallback*). Wywołanie zwrotne przechwytu próbuje obsłużyć wiadomość przechwytującą bezpośrednio ze sterownika VID. W przypadku gdy bezpośrednie obsłużenie nie jest możliwe, zostaje zasygnalizowane zdarzenie synchronizacyjne, które umożliwia zakończenie pętli rozsyłania i umożliwia jednemu z wątków roboczych procesu roboczego maszyny wirtualnej (VMWP) wysłanie przechwytu w trybie użytkownika.

Przełączniki kontekstu, gdy włączony jest planista partycji głównej (*root*), są bardziej kosztowne w porównaniu z innymi implementacjami planisty hipernadzorcy. Kiedy system przełącza się na przykład między dwoma procesorami wirtualnymi gości, zawsze musi wygenerować dwa wyjścia do partycji głównych (*root*). Zintegrowany planista traktuje wątki głównego wirtualnego procesora hipernadzorcy i wątki wirtualnych procesorów gości bardzo różnie (są one jednak wewnętrznie odzwierciedlane przez tę samą strukturę danych *TH_THREAD*):

- Tylko wątek głównego (*root*) procesora wirtualnego może przenieść wątek procesora wirtualnego gościa na swój fizyczny procesor. Wątek głównego procesora wirtualnego ma pierwszeństwo przed każdym procesorem wirtualnym gościa, który jest uruchomiony lub jest rozsyłany. Jeśli główny (*root*) procesor wirtualny nie jest zablokowany, zintegrowany planista stara się jak najlepiej przełączyć kontekst do wątku głównego procesora wirtualnego tak szybko, jak to możliwe.

- Wątek procesora wirtualnego gościa ma dwa zestawy stanów: *stany wewnętrzne wątku* i *stany wątku głównego*. Stany wątku głównej partycji (*root*) odzwierciedlają stany wątku VP-dispatch, które hipernadzorca przekazuje do partycji głównej. Zintegrowany planista utrzymuje te stany dla każdego wątku procesora wirtualnego gościa, aby wiedzieć, kiedy wysłać do partycji głównej (*root*) sygnał budzenia dla odpowiedniego wątku VP-dispatch.

Tylko główny procesor wirtualny może zainicjalizować rozsyłanie procesora wirtualnego gościa dla swojego procesora. Może to zrobić albo w odpowiedzi na hiperwywołania *HvDispatchVp* (w tej sytuacji mówimy, że hipernadzorca przetwarza „pracę zewnętrzną"), albo w odpowiedzi na każde inne hiperwywołanie, które wymaga wysłania synchronicznego żądania do docelowego procesora wirtualnego gościa (to właśnie jest określane jako „praca wewnętrzna"). Jeśli procesor wirtualny gościa ostatnio działał na bieżącym procesorze fizycznym, planista może od razu wysłać wątek procesora wirtualnego gościa. W przeciwnym razie planista musi wysłać żądanie opróżnienia do procesora, na którym procesor wirtualny gościa ostatnio działał, i czekać, aż zdalny procesor wyczyści kontekst procesora wirtualnego. Ten drugi przypadek jest definiowany jako „migracja" i jest sytuacją, którą hipernadzorca musi śledzić (poprzez stany wewnętrzne wątku i stany partycji głównej, które nie są tu opisane).

Eksperyment: praktyczne korzystanie z głównego planisty

Planista NT decyduje, kiedy wybrać i uruchomić wirtualny procesor należący do maszyny wirtualnej i na jak długo. Ten eksperyment pokazuje to, o czym mówiliśmy wcześniej: Wszystkie wątki VP-dispatch są wykonywane w kontekście procesu VMMEM, utworzonego przez sterownik VID. Do przeprowadzenia eksperymentu potrzebna jest stacja robocza z zainstalowaną co najmniej aktualizacją systemu Windows 10 z kwietnia 2018 (RS4) wraz z włączoną rolą Hyper-V oraz gotowa do użycia maszyna wirtualna z zainstalowanym dowolnym systemem operacyjnym. Procedura tworzenia maszyny wirtualnej została szczegółowo wyjaśniona tutaj: *https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/quick-create-virtual-machine*.

W pierwszej kolejności sprawdź, czy włączony jest główny planista. Szczegóły dotyczące tej procedury znajdują się w eksperymencie „Kontrolowanie typu planisty hipernadzorcy" we wcześniejszej części tego rozdziału. Maszyna wirtualna używana do testów powinna być wyłączona.

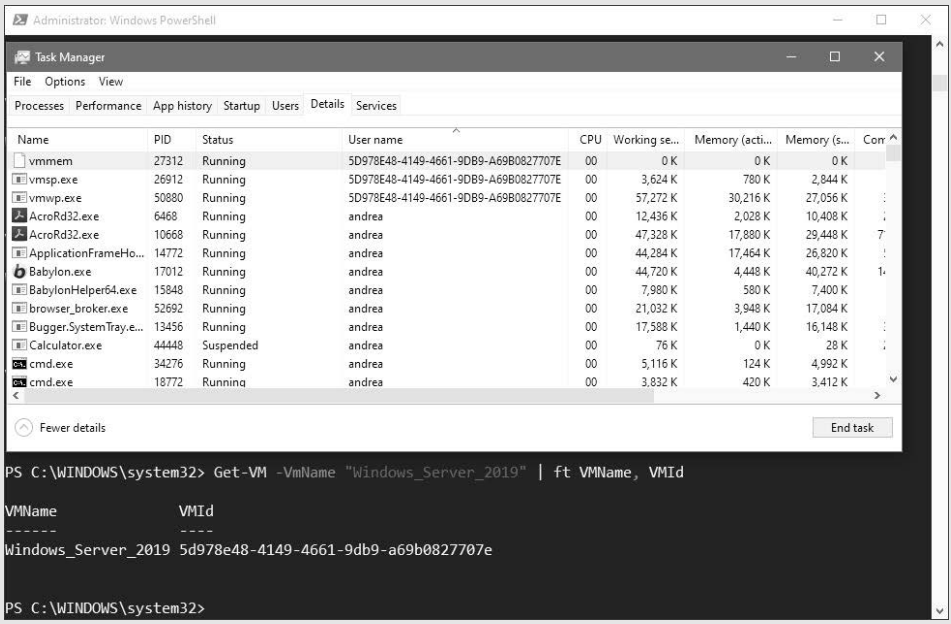
Otwórz menedżera zadań, klikając prawym przyciskiem myszy na pasku zadań i wybierając opcję *Task Manager* (*Menedżer zadań*), kliknij arkusz *Details* (*Szczegóły*) i sprawdź, ile procesów VMMEM jest obecnie aktywnych. W przypadku gdy nie jest uruchomiona żadna maszyna wirtualna, nie powinno być żadnego z nich; w przypadku gdy zainstalowana jest rola Windows Defender Application Guard (WDAG), może istnieć instancja procesu VMMEM, która hostuje wstępnie załadowany kontener WDAG (ten rodzaj maszyny wirtualnej opisany jest w późniejszym punkcie „Maszyny wirtualne wspierane przez VA"). W przypadku istnienia wystąpienia procesu VMMEM zwróć uwagę na jej *Process ID* (*Identyfikator PID*).

Otwórz menedżera funkcji Hyper-V, wpisując **`Hyper-V Manager`** w polu wyszukiwania na pasku zadań, i uruchom swoją maszynę wirtualną. Po uruchomieniu maszyny wirtualnej i pomyślnym uruchomieniu systemu operacyjnego gościa przełącz się z powrotem do menedżera zadań i wyszukaj nowy proces VMMEM. Jeśli klikniesz nowy proces VMMEM i rozwiniesz kolumnę *User Name* (*Nazwa użytkownika*), możesz zobaczyć, że proces został skojarzony z tokenem należącym do użytkownika o nazwie takiej samej jak GUID maszyny wirtualnej.

Możesz uzyskać GUID Maszyny wirtualnej, wykonując następujące polecenie w administracyjnym oknie PowerShell (zastąp wyrażenie *<NazwaVM>* nazwą swojej maszyny wirtualnej):

```
Get-VM -VmName "<NazwaVM>" | ft VMName, VmId
```

ID maszyny wirtualnej i nazwa użytkownika procesu VMMEM powinny być takie same, jak na poniższym rysunku.



Zainstaluj eksplorator procesów (*Process Explorer*), pobierając go ze strony *https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer*, i uruchom go jako administrator. Wyszukaj PID właściwego procesu VMMEM zidentyfikowanego w poprzednim kroku (27312 w przykładzie), kliknij go prawym przyciskiem myszy i wybierz *Suspend* (zawieś proces). W zakładce *CPU* procesu *VMMEM* zamiast prawidłowego czasu pracy procesora powinien się pojawić napis *Suspended* (proces zawieszony).

Jeśli wrócisz do maszyny wirtualnej, zauważysz, że nie reaguje ona na żadne polecenia i całkowicie się zatrzymała. Dzieje się tak dlatego, że zawieszony został proces obsługujący wątki rozsyłania wszystkich procesorów wirtualnych należących do maszyny wirtualnej. To uniemożliwiło jądru NT zaplanowanie tych wątków, co nie pozwoli sterownikowi WinHvr na wyemitowanie potrzebnego hiperwywołania *HvDispatchVp* używanego do wznowienia wykonywania procesora wirtualnego.

Jeśli klikniesz prawym przyciskiem myszy na zawieszony VMMEM i wybierzesz opcję *Resume* (przywróć), maszyna wirtualna wznowi swoje wykonywanie i będzie działać poprawnie.

Hiperwywołania i TLFS hipernadzorcy

Hiperwywołania zapewniają systemowi operacyjnemu działającemu w partycji głównej lub partycji podrzędnej mechanizm żądania usług od hipernadzorcy. Hiperwywołania mają dobrze zdefiniowany zestaw parametrów wejściowych i wyjściowych. Specyfikacja funkcjonalna najwyższego poziomu hipernadzorcy TLFS (ang. *Hipervisor Top Level Functional Specification*) jest dostępna online (*https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs*); definiuje ona różne konwencje wywoływania używane podczas określania tych parametrów. Ponadto wymienia wszystkie publicznie dostępne funkcje hipernadzorcy, właściwości partycji, hipernadzorcę i interfejsy VSM.

Hiperwywołania są dostępne dzięki zależnemu od platformy kodowi operacyjnemu (VMCALL dla systemów Intel, VMMCALL dla AMD, HVC dla ARM64), które po wywołaniu zawsze powodują VM_EXIT do hipernadzorcy. VM_EXIT są zdarzeniami, które skutkują ponownym uruchomieniem hipernadzorcy w celu wykonania własnego kodu na poziomie uprawnień hipernadzorcy, który jest wyższy niż jakiekolwiek inne oprogramowanie uruchomione w systemie (z wyjątkiem kontekstu SMM firmware'u), podczas gdy procesor wirtualny jest zawieszony. Zdarzenia VM_EXIT mogą być generowane z różnych powodów. W specyficznej dla platformy VMCS (lub VMCB) nieprzezroczystej strukturze danych sprzęt utrzymuje indeks określający powód wyjścia VM_EXIT. Hipernadzorca pobiera ten indeks, a w przypadku wyjścia spowodowanego przez hiperwywołanie odczytuje wartość wejściową hiperwywołania określoną przez wywołującego (zazwyczaj z rejestru ogólnego przeznaczenia procesora — RCX w przypadku 64-bitowych systemów Intela i AMD). Wartość wejściowa hiperwywołania (zobacz rysunek 9.16) jest 64-bitową wartością określającą kod hiperwywołania, jego właściwości i konwencję wywołania użytą dla hiperwywołania. Dostępne są trzy rodzaje konwencji wywoływania:

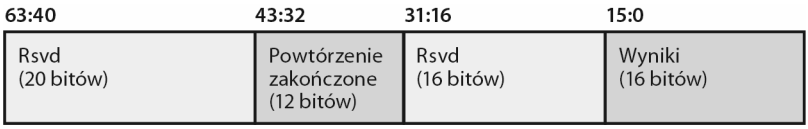
- **Standardowe hiperwywołania** — przechowują parametry wejściowe i wyjściowe na 8-bajtowych przylegających adresach fizycznych gości (GPA). System operacyjny przekazuje te dwa adresy przez rejestry ogólnego przeznaczenia (RDX i R8 w systemach 64-bitowych Intela i AMD).
- **Szybkie hiperwywołania** — zwykle nie dopuszczają parametrów wyjściowych i wykorzystują dwa rejestry ogólnego przeznaczenia używane w standardowych hiperwywołaniach do przekazywania hipernadzorcy jedynie parametrów wejściowych (o rozmiarze do 16 bajtów).
- **Rozszerzone szybkie hiperwywołania** (lub szybkie hiperwywołania XMM) — podobne do szybkich hiperwywołań, ale wykorzystują dodatkowe sześć rejestrów zmiennoprzecinkowych, aby umożliwić wywołującemu przekazanie parametrów wejściowych o rozmiarze do 112 bajtów.

63:60	59:48	47:44	43:32	31:27	26:17	16	15:0
RsvdZ (4 bity)	Indeks początkowy powtórzeń (12 bitów)	RsvdZ (4 bity)	Licznik powtórzeń (12 bitów)	RsvdZ (5 bitów)	Zmienny rozmiar nagłówka (9 bitów)	Szybkie (hiperwywołanie) (1 bit)	Kod wywołania (16 bitów)

RYSUNEK 9.16. *Wartość wejściowa hiperwywołania (z TLFS hipernadzorcy)*

Istnieją dwie klasy hiperwywołań: prosta (ang. *simple*) i powtarzalna (ang. *rep*) — skrót od ang. *repeat*, czyli powtarzać. *Proste* hiperwywołanie wykonuje pojedynczą operację i ma ustalony rozmiar zestawu parametrów wejściowych i wyjściowych. Hiperwywołanie powtarzalne *rep* działa jak seria prostych hiperwywołań. Gdy wywołujący początkowo wywołuje hiperwywołanie powtarzalne, to określa ilość powtórzeń (ang. *rep count*), która wskazuje na liczbę elementów listy parametrów wejściowych lub wyjściowych. Wywołujący określa również indeks początkowy powtórzeń, który wskazuje na następny element wejściowy lub wyjściowy, jaki powinien zostać skonsumowany.

Wszystkie hiperwywołania zwracają inną 64-bitową wartość zwaną *wartością wyniku hiperwywołania* (patrz rysunek 9.17). Ogólnie rzecz biorąc, wartość wyniku opisuje rezultat operacji, a w przypadku hiperwywołań powtarzalnych, całkowitą liczbę zakończonych powtórzeń.



RYSUNEK 9.17. *Wartość wyniku hiperwywołania (z TLFS hipernadzorcy)*

Wykonanie hiperwywołania może zająć trochę czasu. Utrzymywanie fizycznego procesora, który nie odbiera przerwań, może być niebezpieczne dla systemu operacyjnego gospodarza. Na przykład Windows posiada mechanizm, który wykrywa, że procesor nie otrzymał swojego przerwania typu „tykanie zegara" (ang. *clock tick*) przez okres czasu dłuższy niż 16 milisekund. Jeśli taki przypadek zostanie wykryty, system zostaje nagle zatrzymany z wyświetlonym niebieskim ekranem śmierci (BSOD — ang. *Blue Screen of Death*). Dlatego hipernadzorca polega na mechanizmie kontynuacji hiperwywołań (*hypercall continuation)* dla niektórych hiperwywołań, w tym wszystkich form hiperwywołań. Jeśli hiperwywołanie nie jest w stanie zakończyć się w wyznaczonym czasie (zwykle 50 mikrosekund), kontrola jest zwracana do wywołującego (poprzez operację zwaną VM_ENTRY), ale wskaźnik instrukcji nie jest przesuwany poza instrukcję, która wywołała hiperwywołanie. Pozwala to na obsługę oczekujących przerwań i zaplanowanie innych wirtualnych procesorów. Kiedy pierwotnie wywołujący wątek wznowi wykonywanie, ponownie wykona instrukcję hiperwywołania i dokona postępu w kierunku zakończenia operacji.

Sterownik zazwyczaj nigdy nie emituje hiperwywołania bezpośrednio przez zależny od platformy kod operacyjny (*opcode*). Zamiast tego wykorzystuje usługi prezentowane przez sterownik interfejsu hipernadzorcy systemu Windows, który jest dostępny w dwóch różnych wersjach:

- ***WinHvr.sys*** — ładowany przy starcie systemu, jeśli system operacyjny jest uruchomiony na partycji głównej i prezentuje hiperwywołania dostępne zarówno na partycji głównej, jak i podrzędnej.
- ***WinHv.sys*** — ładowany tylko wtedy, gdy system działa na partycji podrzędnej. Udostępnia hiperwywołania dostępne tylko w partycji podrzędnej.

Programy i struktury danych eksportowane przez sterownik interfejsu hipernadzorcy Windows są szeroko wykorzystywane przez stos wirtualizacji, a zwłaszcza przez sterownik VID, który — jak już wspominaliśmy — pełni kluczową rolę w funkcjonalności całej platformy Hyper-V.

Przechwyty

Partycja główna powinna być w stanie stworzyć środowisko wirtualne pozwalające na uruchomienie niezmodyfikowanego systemu operacyjnego gościa, który został napisany do wykonywania na fizycznym sprzęcie, w partycji gościa hipernadzorcy. Tacy dotychczasowi goście mogą próbować uzyskać dostęp do urządzeń fizycznych, które nie istnieją w partycji hipernadzorcy (na przykład poprzez dostęp do określonych portów we/wy lub zapisywanie do określonych MSR-ów). W takich przypadkach hipernadzorca udostępnia funkcję *przechwytów hosta*; gdy VP maszyny wirtualnej gościa wykonuje pewne instrukcje lub generuje pewne wyjątki, autoryzowana partycja główna (*root*) może przechwycić zdarzenie i zmienić efekt przechwyconej instrukcji tak, aby dla partycji podrzędnej odzwierciedlała ona oczekiwane zachowanie w sprzęcie fizycznym.

Gdy zdarzenie przechwytujące wystąpi w partycji podrzędnej, jej procesor wirtualny zostaje zawieszony, a *wiadomość przechwytująca* jest wysyłana od hipernadzorcy do partycji głównej przez SynIC (więcej szczegółów w dalszej części rozdziału). Wiadomość jest odbierana dzięki syntetycznemu programowi obsługi przerwania ISR (ang. *Interrupt Service Routine*) hipernadzorcy, którą jądro NT instaluje podczas fazy 0 swojego startu tylko w przypadku, gdy system jest „oświecony" i działa pod hipernadzorcą (więcej szczegółów w rozdziale 12.). Syntetyczny program obsługi przerwania ISR hipernadzorcy (*KiHvInterrupt*), instalowany zwykle na wektorze 0x30, przekazuje swoje wykonanie do zewnętrznego wywołania zwrotnego, które sterownik VID zarejestrował podczas swojego startu (poprzez eksponowany interfejs API jądra NT *HvlRegisterInterruptCallback*).

Sterownik VID jest sterownikiem przechwytującym, co oznacza, że jest w stanie zarejestrować przechwytywanie hosta w hipernadzorcy i w ten sposób otrzymuje wszystkie zdarzenia przechwytujące, które występują na partycjach podrzędnych. Po zainicjalizowaniu partycji proces roboczy maszyny wirtualnej rejestruje przechwyty dla różnych składników stosu wirtualizacji. (Na przykład wirtualna płyta główna rejestruje przechwyty we/wy dla każdego z wirtualnych portów COM maszyny wirtualnej). Wysyła on IOCTL do sterownika VID, który używa hiperwywołania *HvInstallIntercept* do zainstalowania przechwytywania na partycji podrzędnej. Kiedy partycja podrzędna podnosi przechwyt, hipernadzorca zawiesza procesor wirtualny i wstrzykuje syntetyczne przerwanie w partycji głównej, które jest zarządzane przez *KiHvInterrupt ISR*. Ta ostatnia procedura przekazuje wykonanie do wywołania zwrotnego przechwytującego zarejestrowany identyfikator wirtualny, które to wywołanie zwrotne zarządza zdarzeniem i ponownie uruchamia procesor wirtualny poprzez wyczyszczenie syntetycznego rejestru wstrzymującego przechwytywanie zawieszonego procesora wirtualnego.

Hipernadzorca obsługuje przechwytywanie następujących zdarzeń w partycji podrzędnej:

- dostęp do portów we/wy (odczyt lub zapis),
- dostęp do MSR-ów procesora wirtualnego (odczyt lub zapis),
- wykonanie instrukcji CPUID,
- wyjątki,
- dostęp do rejestrów ogólnego przeznaczenia,
- hiperwywołania.

Syntetyczny kontroler przerwań (SynIC)

Hipernadzorca wirtualizuje przerwania i wyjątki zarówno dla partycji głównej, jak i partycji gościa poprzez syntetyczny kontroler przerwań (SynIC), który jest rozszerzeniem zwirtualizowanego lokalnego APIC (więcej szczegółów na temat APIC można znaleźć w podręczniku programisty Intel lub AMD). SynIC jest odpowiedzialny za wysyłanie wirtualnych przerwań do procesorów wirtualnych (VP). Przerwania dostarczane do partycji dzielą się na dwie kategorie: *zewnętrzne* i *syntetyczne* (zwane też wewnętrznymi lub po prostu przerwaniami wirtualnymi). Przerwania zewnętrzne pochodzą z innych partycji lub urządzeń; przerwania syntetyczne pochodzą z samego hipernadzorcy i są kierowane do procesora wirtualnego partycji.

Kiedy tworzony jest procesor wirtualny w partycji, hipernadzorca tworzy i inicjalizuje SynIC dla każdego obsługiwanego VTL-a. Następnie uruchamia on SynIC VTL 0, co oznacza, że umożliwia wirtualizację APIC fizycznego CPU w sprzętowej strukturze danych VMCS (lub VMCB). Hipernadzorca obsługuje trzy rodzaje wirtualizacji APIC podczas obsługi *zewnętrznych* przerwań sprzętowych:

- W standardowej konfiguracji APIC jest wirtualizowany poprzez sprzętową obsługę wtrysku zdarzeń. Oznacza to, że za każdym razem, gdy partycja uzyskuje dostęp do lokalnych rejestrów APIC procesora wirtualnego, portów we/wy lub MSR (w przypadku x2APIC), wytwarza VMEXIT, powodując, że kody hipernadzorcy wysyłają przerwanie poprzez SynIC, który ostatecznie „wstrzykuje" zdarzenie do właściwego wirtualnego procesora gościa poprzez manipulację nieprzezroczystymi polami VMCS/VMCB (po tym jak przejdzie przez logikę podobną do fizycznego kontrolera APIC, która określa, czy przerwanie może być dostarczone).
- Tryb emulacji APIC działa podobnie do standardowej konfiguracji. Każde fizyczne przerwanie wysyłane przez sprzęt (zwykle przez IOAPIC) nadal powoduje VMEXIT, ale hipernadzorca nie musi wstrzykiwać żadnego zdarzenia. Zamiast tego manipuluje stroną *virtual-APIC* używaną przez procesor do wirtualizacji pewnych dostępów do rejestrów APIC. Kiedy hipernadzorca chce wstrzyknąć zdarzenie, po prostu manipuluje niektórymi wirtualnymi rejestrami zmapowanymi na stronie virtual-APIC. Zdarzenie jest dostarczane przez sprzęt w momencie wystąpienia VMENTRY. Jednocześnie, jeśli wirtualny procesor gościa manipuluje pewnymi częściami swojego lokalnego APIC, nie wytwarza żadnego VMEXIT, ale modyfikacja zostanie zapisana na stronie wirtualnego kontrolera APIC.
- Delegowane przerwania pozwalają na dostarczenie pewnych rodzajów zewnętrznych przerwań bezpośrednio w partycji gościa bez wytwarzania VMEXIT. Pozwala to na mapowanie urządzeń o dostępie bezpośrednim bezpośrednio w partycji podrzędnej bez ponoszenia kar za wydajność spowodowaną przez instrukcje VMEXIT. Procesor fizyczny przetwarza wirtualne przerwania, bezpośrednio zapisując je jako oczekujące na stronie wirtualnego kontrolera APIC. (Więcej szczegółów można znaleźć w podręczniku programisty Intel lub AMD).

Gdy hipernadzorca uruchamia procesor, zwykle inicjalizuje moduł syntetycznego kontrolera przerwań procesora fizycznego (odzwierciedlany przez strukturę danych *CPU_PLS*). Moduł SynIC procesora fizycznego jest tablicą deskryptorów przerwania, które tworzą połączenie między przerwaniem fizycznym a przerwaniem wirtualnym. Deskryptor przerwania hipernadzorcy (wpis w tablicy deskryptorów przerwań (IDT — ang. *Interrupt Descriptor Table*)), jak pokazano to na rysunku 9.18, zawiera dane potrzebne do prawidłowego wysłania przerwania przez moduł SynIC, w szczególności podmiot, do którego przerwanie jest dostarczane (partycja, hipernadzorca, fałszywe przerwanie), docelowy procesor wirtualny (*root*, podrzędny, wiele procesorów wirtualnych lub przerwanie syntetyczne), wektor przerwania, docelowy VTL i kilka innych cech przerwania.

RYSUNEK 9.18. *Deskryptor fizycznego przerwania hipernadzorcy*

W domyślnych konfiguracjach wszystkie przerwania są dostarczane do partycji głównej w VTL 0 lub do samego hipernadzorcy (w drugim przypadku wpis przerwania to *Hypervisor Reserved* — hipernadzorca zarezerwowany). Przerwania zewnętrzne mogą być dostarczane do partycji gościa tylko wtedy, gdy urządzenie bezpośredniego dostępu jest mapowane do partycji podrzędnej; dobrym przykładem są tu urządzenia NVMe.

Za każdym razem, gdy wątek wspierający procesor wirtualny jest wybierany do wykonania, hipernadzorca sprawdza, czy musi zostać dostarczone jedno przerwanie syntetyczne lub większa ich liczba. Jak już wcześniej wspomniano, syntetyczne przerwania nie są generowane przez żaden sprzęt; są one zazwyczaj generowane z poziomu samego hipernadzorcy (pod pewnymi warunkami), a nadal są zarządzane przez SynIC, który jest w stanie wstrzyknąć wirtualne przerwanie do właściwego procesora wirtualnego. Mimo że są one szeroko wykorzystywane przez jądro NT (dobrym przykładem jest „oświecony" czasomierz zegara), syntetyczne przerwania mają fundamentalne znaczenie dla wirtualnego trybu bezpiecznego VSM. Omawiamy je w punkcie „Bezpieczne jądro" w dalszej części tego rozdziału.

Partycja główna może wysłać niestandardowe wirtualne przerwanie do partycji podrzędnej, używając hiperwywołania *HvAssertVirtualInterrupt* (udokumentowanego w TLFS).

Komunikacja między partycjami

Syntetyczny kontroler przerwań ma również ważną rolę polegającą na zapewnianiu maszynom wirtualnym udogodnień w komunikacji pomiędzy partycjami. Hipernadzorca udostępnia dwa podstawowe mechanizmy komunikacji jednej partycji z drugą: komunikaty i zdarzenia. W obu przypadkach powiadomienia są wysyłane do docelowego procesora wirtualnego za pomocą przerwań syntetycznych. Komunikaty i zdarzenia są wysyłane z partycji źródłowej do docelowej poprzez wstępnie przydzielone *połączenie*, które jest powiązane z *portem* docelowym.

Jednym z najważniejszych komponentów, który wykorzystuje usługi komunikacji między partycjami dostarczane przez SynIC, jest VMBusv (architektura VMBus jest omówiona w podrozdziale „Stos wirtualizacji" w dalszej części tego rozdziału). Sterownik główny (*root*) VMBus (*Vmbusr.sys*) w partycji głównej (*root*) przydziela identyfikator portu (porty są identyfikowane przez 32-bitowy identyfikator) i tworzy port w partycji podrzędnej, emitując hiperwywołanie *HvCreatePort* poprzez usługi dostarczane przez sterownik WinHv.

Port jest przydzielany w hipernadzorcy z puli pamięci odbiorcy. Kiedy port jest tworzony, hipernadzorca przydziela 16 buforów wiadomości z pamięci portu. Bufory komunikatów są utrzymywane w kolejce związanej z syntetycznym źródłem przerwań (SINT — ang. *Synthetic Interrupt Source*) w syntetycznym kontrolerze przerwań (SynIC) wirtualnego procesora. Hipernadzorca eksponuje 16 źródeł przerwań, dzięki czemu sterownik główny VMBus może zarządzać maksymalnie 16 kolejkami komunikatów. Komunikat syntetyczny ma stały rozmiar 256 bajtów i może przesłać tylko 240 bajtów (16 bajtów jest używanych jako nagłówek). Wywoływacz hiperwywołania *HvCreatePort* określa, który procesor wirtualny i SINT ma być docelowy.

Aby prawidłowo odbierać komunikaty, sterownik WinHv przydziela syntetyczną stronę komunikatów o przerwaniu (SIMP — ang. *Synthetic Interrupt Message Page*), która jest następnie współdzielona z hipernadzorcą. Kiedy komunikat oczekuje w kolejce dla partycji docelowej, hipernadzorca kopiuje komunikat ze swojej wewnętrznej kolejki do slotu SIMP odpowiadającego właściwemu SINT. Następnie sterownik VMBus *root* tworzy *połączenie*, które kojarzy port otwarty w podrzędnej maszynie wirtualnej z maszyną nadrzędną, poprzez hiperwywołanie *HvConnectPort*. Po włączeniu przez wirtualną maszynę podrzędną odbioru przerwań syntetycznych w odpowiednim slocie SINT można rozpocząć komunikację; nadawca może wysłać wiadomość do klienta, określając docelowy ID portu i emitując hiperwywołanie *HvPostMessage*. Hipernadzorca wstrzykuje syntetyczne przerwanie do docelowego procesora wirtualnego, który może odczytać ze strony komunikatu (SIMP) treść wiadomości.

Hipernadzorca obsługuje porty i połączenia trzech typów:

- **Porty komunikatów** — przesyłają 240-bajtowe komunikaty z i do partycji. Port komunikatów jest związany z pojedynczym SINT w partycji nadrzędnej i podrzędnej. Wiadomości będą dostarczane w kolejności przez kolejkę wiadomości pojedynczego portu. Ta cecha sprawia, że komunikaty są idealne do ustawiania i wyłączania kanałów VMBus (dalsze szczegóły są podane w podrozdziale „Stos wirtualizacji" w dalszej części tego rozdziału).
- **Porty zdarzeń** — odbierają proste przerwania związane z zestawem flag, ustawianych przez hipernadzorcę, gdy przeciwny punkt końcowy wykona hiperwywołanie *HvSignalEvent*. Ten rodzaj portu jest zwykle używany jako mechanizm synchronizacji. VMBus, na przykład, używa portu zdarzeń do powiadomienia, że wiadomość została umieszczona na buforze pierścieniowym opisanym przez określony kanał. Kiedy przerwanie zdarzenia jest dostarczane do partycji docelowej, odbiorca wie dokładnie, do którego kanału skierowane jest przerwanie dzięki fladze związanej ze zdarzeniem.
- **Porty monitorujące** — optymalizacja dla portu *Zdarzenie* (*Event*). Powodowanie wykonania instrukcji VMEXIT i przełączanie kontekstu maszyny wirtualnej dla każdego pojedynczego wywołania *HvSignalEvent* jest kosztowną operacją. Porty monitorujące są tworzone poprzez przydzielenie współdzielonej strony (pomiędzy hipernadzorcą a partycją), która zawiera strukturę danych wskazującą, który port zdarzeń jest związany z konkretną monitorowaną flagą powiadomienia (bit na stronie). W ten sposób, gdy partycja źródłowa chce wysłać przerwanie synchronizacji, może po prostu ustawić odpowiednią flagę na stronie współdzielonej. Prędzej czy później hipernadzorca zauważy bit ustawiony na stronie współdzielonej i wywoła przerwanie do portu zdarzeń.

API platformy hipernadzorcy systemu Windows i partycje EXO

System Windows w coraz większym stopniu wykorzystuje hipernadzorcę platformy Hyper-V do zapewnienia funkcjonalności nie tylko związanej z uruchamianiem tradycyjnych maszyn wirtualnych. W szczególności, jak omówimy w drugiej części tego rozdziału, VSM, ważny składnik zabezpieczeń nowoczesnych wersji systemu Windows, wykorzystuje hipernadzorcę do wymuszenia wyższego poziomu izolacji dla funkcji, które zapewniają krytyczne usługi systemowe lub obsługują tajne dane, takie jak hasła. Włączenie tych funkcji wymaga, aby hipernadzorca był domyślnie uruchomiony na maszynie.

Zewnętrzne produkty wirtualizacyjne, takie jak VMware, Qemu, VirtualBox, Android Emulator i wiele innych, używają rozszerzeń wirtualizacji dostarczanych przez sprzęt do budowania własnych hipernadzorców, co jest potrzebne, by umożliwić ich poprawne działanie. Jest to oczywiście niezgodne z platformą Hyper-V, która uruchamia swojego hipernadzorcę przed uruchomieniem jądra Windows na partycji głównej (hipernadzorca systemu Windows jest hipernadzorcą natywnym, czyli bare-metalowym).

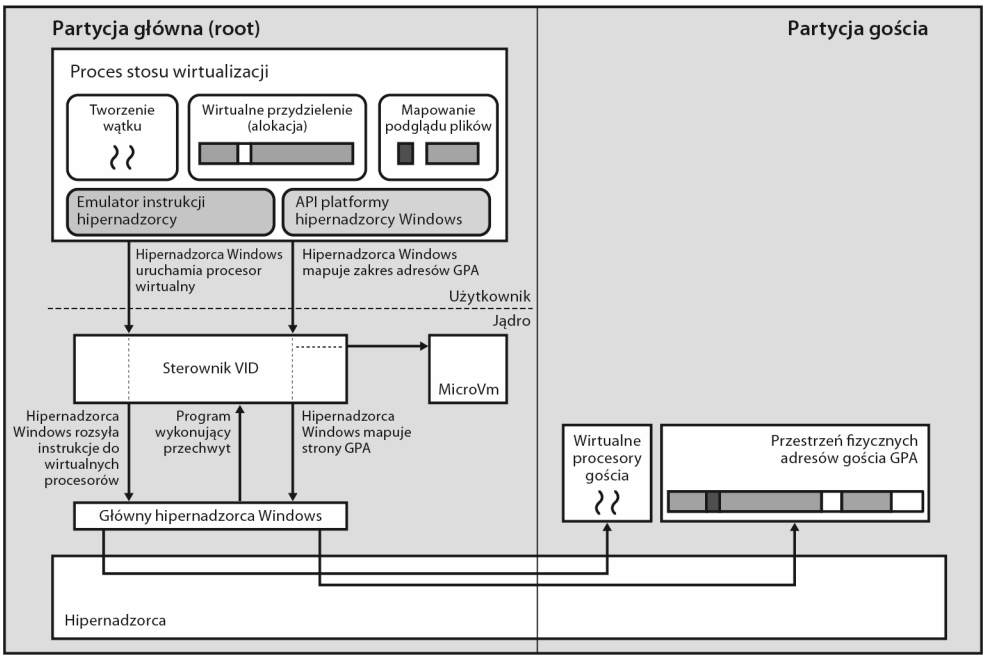
Podobnie jak w przypadku platformy Hyper-V, zewnętrzne rozwiązania wirtualizacyjne również składają się z hipernadzorcy, który zapewnia ogólne niskopoziomowe abstrakcje dla wykonania instrukcji procesora i zarządzania pamięcią maszyny wirtualnej, oraz stosu wirtualizacji, który odnosi się do komponentów rozwiązania wirtualizacyjnego, zapewniających emulowane środowisko dla maszyny wirtualnej (takie jak płyta główna, oprogramowanie układowe firmware, kontrolery pamięci masowej, urządzenia i tak dalej).

API platformy hipernadzorcy Windows (ang. *Windows Hypervisor Platform API*), którego dokumentacja znajduje się pod adresem *https://docs.microsoft.com/en-us/virtualization/api/*, ma na celu umożliwienie uruchamiania rozwiązań wirtualizacyjnych innych firm na hipernadzorcy Windows. W szczególności produkt wirtualizacyjny innej firmy powinien być w stanie tworzyć, usuwać, uruchamiać i zatrzymywać maszyny wirtualne o cechach (oprogramowanie układowe firmware, emulowane urządzenia, kontrolery pamięci masowej) zdefiniowanych przez własny stos wirtualizacji. Stos wirtualizacji innej firmy, wraz z interfejsami zarządzania, nadal działa w systemie Windows na partycji głównej, co pozwala na niezmienione korzystanie z jego maszyn wirtualnych przez ich klienta.

Jak pokazano na rysunku 9.19, wszystkie interfejsy API platformy hipernadzorcy Windows działają w trybie użytkownika i są zaimplementowane na warstwie sterownika VID i WinHvr w dwóch bibliotekach: *WinHvPlatform.dll* i *WinHvEmulation.dll* (ta ostatnia implementuje emulator instrukcji dla MMIO).

Aplikacja trybu użytkownika, która chce utworzyć maszynę wirtualną i jej odpowiednie procesory wirtualne, powinna zwykle wykonać następujące czynności:

1. Utworzyć partycję w bibliotece VID (*Vid.dll*) za pomocą interfejsu API *WHvCreatePartition*.
2. Skonfigurować różne wewnętrzne właściwości partycji — liczbę procesorów wirtualnych, tryb emulacji APIC, rodzaj żądanych instrukcji VMEXIT itd. — używając API *WHvSetPartitionProperty*.
3. Utworzyć partycję w sterowniku VID i hipernadzorcy za pomocą interfejsu API *WHvSetupPartition*. (Ten rodzaj partycji w hipernadzorcy nazywany jest partycją EXO, jak opisano poniżej). API tworzy również procesory wirtualne partycji, które są tworzone w stanie zawieszenia.



RYSUNEK 9.19. *Architektura API platformy hipernadzorcy Windows*

4. Utworzyć odpowiedni wirtualny procesor (lub procesory) w bibliotece VID za pomocą interfejsu API *WHvCreateVirtualProcessor*. Ten krok jest ważny, ponieważ API ustawia i mapuje bufor komunikatów w aplikacji trybu użytkownika, który jest używany do asynchronicznej komunikacji z hipernadzorcą i wątkiem uruchamiającym wirtualne procesory.
5. Przydzielić przestrzeń adresową partycji, rezerwując duży zakres pamięci wirtualnej za pomocą klasycznej funkcji *VirtualAlloc* (więcej szczegółów w rozdziale 5. w części I) i zmapuj ją w hipernadzorcy za pomocą interfejsu API *WHvMapGpaRange*. Drobnoziarnista ochrona pamięci fizycznej gościa może być określona podczas przydzielenia pamięci fizycznej gościa w wirtualnej przestrzeni adresowej gościa poprzez zadeklarowanie różnych zakresów zarezerwowanej pamięci wirtualnej.
6. Utworzyć tablice stron i skopiuj początkowy kod oprogramowania układowego firmware'u w pamięci zadeklarowanej (ang. *commit*).
7. Ustawić początkową zawartość rejestrów wirtualnego procesora za pomocą interfejsu API *WHvSetVirtualProcessorRegisters*.
8. Uruchomić wirtualny procesor, wywołując funkcję *WHvRunVirtualProcessor* blokującą API. Funkcja jest zwracana tylko wtedy, gdy kod gościa wykona operację wymagającą obsługi w stosie wirtualizacji (VMEXIT w hipernadzorcy został wyraźnie zażądany do zarządzania przez zewnętrzny stos wirtualizacji) lub z powodu zewnętrznego żądania (jak na przykład niszczenie procesora wirtualnego).

Interfejsy API platformy hipernadzorcy Windows są zwykle w stanie wywoływać usługi w hipernadzorcy, wysyłając różne IOCTL do obiektu urządzenia *\Device\VidExo*, który jest tworzony przez sterownik VID w czasie inicjalizacji, tylko wtedy gdy wartość rejestru *HKLM\System\CurrentControlSet\Services\Vid\Parameters\ExoDeviceEnabled* jest ustawiona na 1. W przeciwnym razie system nie włącza żadnej obsługi interfejsów API hipernadzorcy.

Niektóre wrażliwe na wydajność interfejsy API platformy hipernadzorcy (dobrym przykładem jest *WHvRunVirtualProcessor*) mogą zamiast tego wywoływać bezpośrednio do hipernadzorcy z trybu użytkownika dzięki stronie *Doorbell* (dzwonek do drzwi), czyli specjalnej nieważnej stronie fizycznej gościa, która po uzyskaniu dostępu zawsze powoduje VMEXIT. API platformy hipernadzorcy Windows uzyskuje adres strony *Doorbell* od sterownika VID. Dokonuje on wpisów do strony *Doorbell* za każdym razem, gdy emituje hiperwywołanie z trybu użytkownika. Błąd jest identyfikowany i traktowany inaczej przez hipernadzorcę dzięki fizycznemu adresowi strony *Doorbell*, który jest oznaczony jako „specjalny" w tablicy stron SLAT. Hipernadzorca odczytuje kod i parametry hiperwywołania z rejestrów procesora wirtualnego, jak w przypadku normalnych hiperwywołań, i ostatecznie przekazuje wykonanie do programu handlera hiperwywołania. Kiedy ta ostatnia kończy swoje wykonanie, hipernadzorca ostatecznie wykonuje VMENTRY, lądując na instrukcji następującej po instrukcji błędnej. Oszczędza to wiele cykli zegarowych wątkowi wspierającemu wirtualny procesor gościa, który nie musi już wchodzić do jądra w celu wyemitowania hiperwywołania. Co więcej, VMCALL i podobne kody operacyjne (*opcodes*) zawsze wymagają uprawnień jądra do wykonania instrukcji.

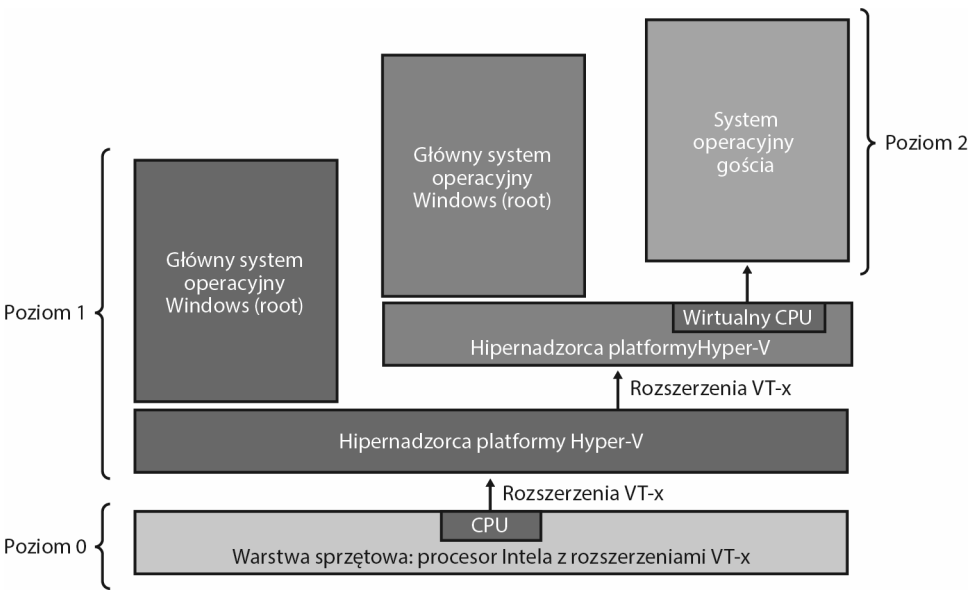
Procesory wirtualne nowej maszyny wirtualnej strony trzeciej są wysyłane za pomocą planisty głównego. Jeśli planista główny jest wyłączony, nie można uruchomić żadnej funkcji API platformy hipernadzorcy. Partycja utworzona w hipernadzorcy jest partycją EXO. Partycje EXO to minimalne partycje, które nie zawierają żadnej syntetycznej funkcjonalności i mają pewne cechy idealne do tworzenia maszyn wirtualnych innych firm:

- Są zawsze typami partycji wspieranych przez wirtualne adresy (ang. *VA-backed*). (Więcej szczegółów na temat maszyn wirtualnych wspieranych przez wirtualne adresy lub mikromaszyn wirtualnych znajduje się w dalszej części rozdziału, w podrozdziale „Stos wirtualizacji"). Procesem obsługującym pamięć partycji jest aplikacja trybu użytkownika, która utworzyła maszynę wirtualną, a nie nowe wystąpienie procesu VMMEM.
- Nie mają one żadnych przywilejów partycji ani nie obsługują żadnego VTL-a (wirtualnego poziomu zaufania) innego niż 0. Wszystkie przywileje partycji klasycznej odnoszą się do funkcjonalności syntetycznej, która jest zwykle prezentowana przez hipernadzorcę stosowi wirtualizacji platformy Hyper-V. Partycje EXO są używane dla stosów wirtualizacji innych firm. Nie potrzebują one funkcjonalności wnoszonej przez którykolwiek z przywilejów partycji klasycznej.
- Ręcznie zarządzają timingiem. Hipernadzorca nie dostarcza żadnego źródła przerwania wirtualnego zegara dla partycji EXO. Odpowiedzialność za jego zapewnienie musi przejąć stos wirtualizacji firm trzecich. Oznacza to, że każda próba odczytania znacznika czasu wirtualnego procesora spowoduje VMEXIT w hipernadzorcy, który skieruje przechwytywanie do wątku trybu użytkownika, który uruchamia wirtualny procesor.

Uwaga. Partycje EXO zawierają inne drobne różnice w porównaniu z klasycznymi partycjami hipernadzorcy. Dla dobra dyskusji te drobne różnice są jednak uznane za nieistotne, więc nie zostały wspomniane w tej książce.

Wirtualizacja zagnieżdżona

Duże serwery i dostawcy chmur czasami potrzebują możliwości uruchamiania kontenerów lub dodatkowych maszyn wirtualnych wewnątrz partycji gościa. Rysunek 9.20 opisuje taki scenariusz: Hipernadzorca, który działa na warstwie sprzętu bare-metalowego, zidentyfikowany jako hipernadzorca L0 (L0 oznacza poziom 0), wykorzystuje rozszerzenia wirtualizacji dostarczane przez sprzęt do utworzenia maszyny wirtualnej gościa. Ponadto hipernadzorca L0 emuluje rozszerzenia wirtualizacji procesora i udostępnia je maszynie wirtualnej gościa (możliwość udostępniania rozszerzeń wirtualizacji jest nazywana *wirtualizacją zagnieżdżoną*). Maszyna wirtualna gościa może zdecydować o uruchomieniu innego wystąpienia hipernadzorcy (który w tym przypadku jest identyfikowany jako hipernadzorca L1, gdzie *L1* oznacza *Level 1*), wykorzystując emulowane rozszerzenia wirtualizacji eksponowane przez hipernadzorcę L0. Hipernadzorca L1 tworzy zagnieżdżoną partycję główną i uruchamia w niej główny (*root*) system operacyjny L2. W ten sam sposób główny (*root*) L2 może współpracować z hipernadzorcą L1, aby uruchomić zagnieżdżoną maszynę wirtualną gościa. Ostatecznie utworzona maszyna wirtualna gościa w tej konfiguracji przyjmuje nazwę *Gość L2*.



RYSUNEK 9.20. *Schemat wirtualizacji zagnieżdżonej*

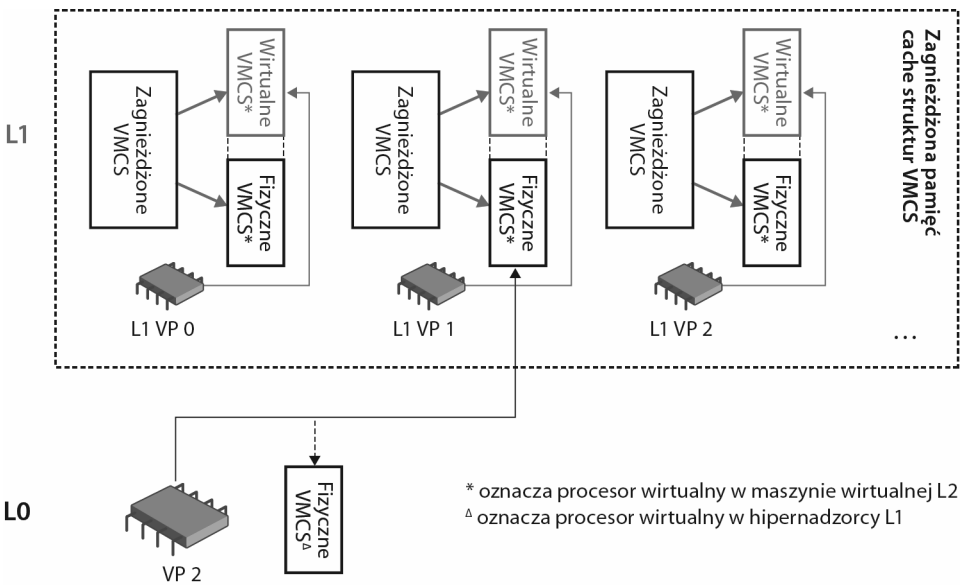
Wirtualizacja zagnieżdżona jest konstrukcją programową: hipernadzorca musi być w stanie emulować i zarządzać rozszerzeniami wirtualizacji. Każda instrukcja wirtualizacji podczas wykonywania przez maszynę wirtualną gościa L1 powoduje VMEXIT do hipernadzorcy L0, który poprzez swój emulator może zrekonstruować instrukcję i wykonać potrzebną pracę, aby ją emulować. W momencie pisania tego tekstu, obsługiwany jest tylko sprzęt Intel i AMD. Możliwość zagnieżdżonej wirtualizacji powinna być jawnie włączona dla maszyny wirtualnej L1; w przeciwnym razie hipernadzorca L0 wstrzykuje ogólny wyjątek ochrony w maszynie wirtualnej, w przypadku gdy instrukcja wirtualizacji jest wykonywana przez system operacyjny gościa.

Na sprzęcie Intela platforma Hyper-V umożliwia działanie zagnieżdżonej wirtualizacji dzięki dwóm głównym koncepcjom:

- emulacji rozszerzeń wirtualizacji VT-x,
- zagnieżdżonej translacji adresów.

Jak omówiono wcześniej w tym rozdziale, w przypadku sprzętu Intela podstawową strukturą danych opisującą maszynę wirtualną jest struktura sterująca maszyną wirtualną (VMCS — ang. *Virtual Machine Control Structure*). Poza standardową fizyczną strukturą VMCS odzwierciedlającą maszynę wirtualną L1, kiedy hipernadzorca L0 tworzy procesor wirtualny należący do partycji obsługującej zagnieżdżoną wirtualizację, przydziela pewne zagnieżdżone struktury danych VMCS (nie mylić z wirtualną strukturą VMCS, która jest inną koncepcją). Zagnieżdżona struktura VMCS jest deskryptorem programowym, który zawiera wszystkie informacje potrzebne hipernadzorcy L0 do uruchomienia i pracy zagnieżdżonego procesora wirtualnego dla partycji L2. Jak pokrótce przedstawiono w punkcie „Uruchamianie hipernadzorcy", kiedy hipernadzorca L1 wykonuje rozruch, wykrywa, czy działa w środowisku zwirtualizowanym, a jeśli tak, to włącza różne zagnieżdżone rozszerzenia „oświecające", takie jak „oświecony" VMCS czy bezpośrednie „spłukanie" wirtualnej zawartości (omówione w dalszej części tej sekcji).

Jak pokazano na rysunku 9.21, dla każdej zagnieżdżonej struktury VMCS hipernadzorca L0 przydziela również wirtualną strukturę VMCS i sprzętową fizyczną strukturę VMCS, dwie podobne struktury danych odzwierciedlające procesor wirtualny uruchamiający maszynę wirtualną L2. Wirtualna struktura VMCS jest ważna, ponieważ pełni kluczową rolę w utrzymywaniu zagnieżdżonych zwirtualizowanych danych. Fizyczna struktura VMCS jest natomiast ładowana przez hipernadzorcę L0 w momencie uruchomienia maszyny wirtualnej L2; dzieje się tak, gdy hipernadzorca L0 przechwyci instrukcję VMLAUNCH wykonywaną przez hipernadzorcę L1.



RYSUNEK 9.21. *Hipernadzorca L0 uruchamiający maszynę wirtualną L2 przez procesor wirtualny 2*

Na przykładowym rysunku hipernadzorca L0 zaplanował procesor wirtualny nr 2 do uruchomienia maszyny wirtualnej L2 zarządzanej przez hipernadzorcę L1 (poprzez zagnieżdżony procesor wirtualny nr 1). Hipernadzorca L1 może operować tylko na danych wirtualizacyjnych replikowanych w wirtualnym VMCS.

Emulacja rozszerzeń wirtualizacji VT-x

Na sprzęcie Intela hipernadzorca L0 obsługuje zarówno „oświeconych", jak i „nieoświeconych" hipernadzorców L1. Jedyną oficjalnie obsługiwaną konfiguracją jest jednak hipernadzorca Hyper-V działający na warstwie platformy Hyper-V.

W „nieoświeconym" hipernadzorcy wszystkie instrukcje VT-x wykonywane w gościu L1 powodują VMEXIT. Po tym jak hipernadzorca L1 przydzielił gościowi fizyczną strukturę VMCS do opisania nowej maszyny wirtualnej L2, zwykle oznacza ją jako aktywną (poprzez instrukcję VMPTRLD na sprzęcie Intela). Hipernadzorca L0 przechwytuje tę operację i kojarzy przydzielony zagnieżdżony VMCS z fizycznym VMCS gościa określonym przez hipernadzorcę L1. Ponadto wypełnia wartości początkowe dla wirtualnej struktury VMCS i ustawia zagnieżdżony VMCS jako aktywny dla bieżącego procesora wirtualnego. (Nie przełącza jednak fizycznej struktury VMCS; kontekstem wykonawczym powinien pozostać hipernadzorca L1). Każdy kolejny odczyt lub zapis do fizycznej struktury VMCS wykonany przez hipernadzorcę L1 jest zawsze przechwytywany przez hipernadzorcę L0 i przekierowywany do wirtualnej struktury VMCS (patrz rysunek 9.21).

Kiedy hipernadzorca L1 uruchamia maszynę wirtualną (poprzez operację zwaną VMENTRY), wykonuje specyficzną instrukcję sprzętową (VMLAUNCH na sprzęcie Intel), która jest przechwytywana przez hipernadzorcę L0. W przypadku „nieoświeconych" scenariuszy hipernadzorca L0 kopiuje wszystkie pola gościa wirtualnej struktury VMCS do innej fizycznej struktury VMCS odzwierciedlającej maszynę wirtualną L2, zapisuje pola hosta, wskazując je na punkty wejścia hipernadzorcy L0, i ustawia maszynę wirtualną L2 jako aktywną (poprzez użycie sprzętowej instrukcji VMPTRLD na platformach Intela). W przypadku gdy hipernadzorca L1 używa translacji adresów drugiego poziomu (EPT w przypadku sprzętu Intel), hipernadzorca L0 następnie tworzy pseudo-tablice na bazie aktualnie aktywnych tablic stron rozszerzonych L1 (więcej szczegółów w dalszej części rozdziału). Na koniec wykonuje faktyczną instrukcję VMENTRY poprzez wykonanie określonej instrukcji sprzętowej. W rezultacie sprzęt wykonuje kod maszyny wirtualnej L2.

Podczas wykonywania maszyny wirtualnej L2 każda operacja powodująca VMEXIT przełącza kontekst wykonawczy z powrotem do hipernadzorcy L0 (zamiast L1). W odpowiedzi na to hipernadzorca L0 wykonuje kolejne VMENTRY na oryginalnym fizycznym VMCS odzwierciedlającym kontekst hipernadzorcy L1, wstrzykując syntetyczne zdarzenie VMEXIT. Hipernadzorca L1 ponownie uruchamia wykonywanie instrukcji i obsługuje przechwycone zdarzenie tak jak w przypadku zwykłych, niezagnieżdżonych instrukcji VMEXIT. Gdy L1 zakończy wewnętrzną obsługę syntetycznego zdarzenia VMEXIT, wykonuje operację VMRESUME, która zostanie ponownie przechwycona przez hipernadzorcę L0 i zarządzana w podobny sposób jak opisana wcześniej początkowa operacja VMENTRY.

Wytwarzanie VMEXIT za każdym razem, gdy hipernadzorca L1 wykonuje instrukcję wirtualizacji, jest kosztowną operacją, która może definitywnie przyczynić się do ogólnego spowolnienia maszyny wirtualnej L2. Aby przezwyciężyć ten problem, hipernadzorca platformy Hyper-V obsługuje „oświeconą" strukturę VMCS, optymalizację, która po włączeniu pozwala hipernadzorcy L1 na ładowanie, czytanie i zapisywanie danych wirtualizacji ze strony pamięci

współdzielonej przez hipernadzorcę L1 i L0 (zamiast fizycznej struktury VMCS). Współdzielona strona jest nazywana „oświeconą" strukturą VMCS. Kiedy hipernadzorca L1 manipuluje danymi wirtualizacji należącymi do maszyny wirtualnej L2, zamiast używać instrukcji sprzętowych, które powodują VMEXIT w hipernadzorcy L0, bezpośrednio odczytuje i zapisuje z „oświeconej" struktury VMCS. To znacznie poprawia wydajność maszyny wirtualnej L2.

W scenariuszach rozszerzeń „oświecających" hipernadzorca L0 przechwytuje tylko operacje VMENTRY i VMEXIT (oraz kilka innych, które nie są istotne dla tej dyskusji). Hipernadzorca L0 zarządza operacjami VMENTRY w podobny sposób jak w scenariuszu „nieoświeconym", ale przed wykonaniem wszystkich opisanych wcześniej czynności kopiuje dane wirtualizacyjne znajdujące się we współdzielonej stronie pamięci „oświeconej" struktury VMCS do wirtualnej struktury VMCS odzwierciedlającej maszynę wirtualną L2.

Uwaga. Warto wspomnieć, że w przypadku scenariuszy „nieoświeconych" hipernadzorców L0 obsługuje inną technikę zapobiegania operacjom VMEXIT podczas zarządzania zagnieżdżonymi danymi wirtualizacyjnymi, zwaną pseudo-strukturą VMCS. Pseudo-struktura VMCS jest optymalizacją sprzętową bardzo podobną do „oświeconej" struktury VMCS.

Zagnieżdżona translacja adresów

Jak wcześniej opisano w punkcie „Fizyczna przestrzeń adresowa partycji", hipernadzorca używa SLAT do dostarczania odizolowanej fizycznej przestrzeni adresowej gościa do maszyny wirtualnej oraz do tłumaczenia GPA na rzeczywiste SPA. Zagnieżdżone maszyny wirtualne wymagałyby kolejnej sprzętowej warstwy translacji umieszczonej na dwóch już istniejących warstwach. W celu obsłużenia zagnieżdżonej wirtualizacji nowa warstwa powinna być w stanie przetłumaczyć L2 GPA na L1 GPA. Ze względu na zwiększoną złożoność elektroniki potrzebnej do zbudowania jednostki zarządzania pamięcią (MMU — ang. *Memory Management Unit*) procesora, która zarządza trzema warstwami translacji, hipernadzorca platformy Hyper-V przyjął inną strategię zapewnienia dodatkowej warstwy translacji adresów, zwaną tablicami pseudo-stron zagnieżdżonych SNPT (ang. *shadow nested page tables*). SNPT wykorzystują technikę podobną do pseudo-stronicowania (ang. *shadow paging*) (patrz poprzedni rozdział) do bezpośredniego tłumaczenia L2 GPA na SPA.

Gdy tworzona jest partycja obsługująca wirtualizację zagnieżdżoną, hipernadzorca L0 przydziela i inicjalizuje pseudo-domenę tablicy stron zagnieżdżonych. Ta struktura danych służy do przechowywania listy tablic pseudo-stron zagnieżdżonych związanych z różnymi maszynami wirtualnymi L2 utworzonymi w partycji. Ponadto przechowuje ona numer generacji aktywnej domeny partycji (omówiony w dalszej części tego rozdziału) oraz statystyki pamięci zagnieżdżonej.

Gdy hipernadzorca L0 wykonuje początkową instrukcję VMENTRY w celu uruchomienia maszyny wirtualnej L2, przydziela tablicę pseudo-stron zagnieżdżonych związaną z tą maszyną wirtualną i inicjalizuje tę tablicę pustymi wartościami (wynikowa fizyczna przestrzeń adresowa jest pusta). Gdy maszyna wirtualna L2 rozpoczyna wykonywanie kodu, natychmiast generuje VMEXIT do hipernadzorcy L0 z powodu błędu zagnieżdżonej strony (naruszenie EPT w sprzęcie Intel). Hipernadzorca L0, zamiast wstrzykiwać błąd do L1, przegląda tablice stron zagnieżdżonych gościa zbudowane przez hipernadzorcę L1. Jeśli znajdzie poprawny wpis dla określonego GPA L2, odczytuje odpowiadający mu GPA L1, tłumaczy go na SPA i tworzy potrzebną hierarchię tablic pseudo-stron zagnieżdżonych, aby zmapować go w maszynie wirtualnej L2. Następnie uzupełnia wpis w tablicy liści (*leaf*) prawidłowym SPA

(hipernadzorca używa dużych stron do mapowania pseudo-stron zagnieżdżonych) i wznawia wykonywanie instrukcji bezpośrednio do maszyny wirtualnej L2, ustawiając opisującą ją zagnieżdżoną strukturę VMCS jako aktywną.

Aby zagnieżdżona translacja adresów działała poprawnie, hipernadzorca L0 powinien być świadomy wszelkich modyfikacji, które mają miejsce w zagnieżdżonych tablicach stron L1; w przeciwnym razie maszyna wirtualna L2 mogłaby działać z nieaktualnymi wpisami. Ta implementacja jest specyficzna dla platformy; zwykli hipernadzorcy chronią zagnieżdżoną tablicę stron L2 dla dostępu tylko do odczytu. W ten sposób mogą być informowani, gdy hipernadzorca L1 modyfikuje tę tablicę. Hipernadzorca platformy Hyper-V przyjmuje jednak inną, inteligentną strategię. Gwarantuje ona, że tablica pseudo-stron zagnieżdżonych opisująca maszynę wirtualną L2 jest zawsze aktualizowana z powodu następujących dwóch przesłanek:

- Gdy hipernadzorca L1 dodaje nowe wpisy w zagnieżdżonej tablicy stron L2, nie wykonuje żadnych innych działań dla zagnieżdżonej maszyny wirtualnej (w hipernadzorcy L0 nie są generowane żadne przechwyty). Wpis w tablicy pseudo-stron zagnieżdżonych jest dodawany tylko wtedy, gdy błąd strony zagnieżdżonej powoduje VMEXIT w hipernadzorcy L0 (scenariusz opisany wcześniej).
- Podobnie jak w przypadku niezagnieżdżonej maszyny wirtualnej, gdy wpis w tablicy stron zagnieżdżonych jest modyfikowany lub usuwany, hipernadzorca powinien zawsze emitować instrukcję opróżnienia bufora TLB w celu prawidłowego unieważnienia sprzętowego bufora TLB. W przypadku zagnieżdżonej wirtualizacji, gdy hipernadzorca L1 emituje instrukcję czyszczenia bufora TLB, L0 przechwytuje żądanie i całkowicie unieważnia tablicę pseudo-stron zagnieżdżonych. Hipernadzorca L0 utrzymuje koncepcję wirtualnego bufora TLB dzięki identyfikatorom generacji przechowywanym zarówno w pseudo strukturze VMCS, jak i pseudo-domenie tablicy stron zagnieżdżonych. (Opisywanie architektury wirtualnego bufora TLB wykracza poza zakres tej książki).

Całkowite unieważnienie tablicy pseudo-stron zagnieżdżonych w przypadku pojedynczej zmiany adresu wydaje się zbędne, ale jest to podyktowane obsługą sprzętową. (Instrukcja INVEPT działająca na sprzęcie Intela nie pozwala określić, który pojedynczy GPA ma zostać usunięty z bufora TLB). W klasycznych maszynach wirtualnych nie jest to problem, ponieważ modyfikacje dokonywane na fizycznej przestrzeni adresowej nie zdarzają się zbyt często. Gdy klasyczna maszyna wirtualna jest uruchamiana, cała jej pamięć jest już przydzielona. (Więcej szczegółów podamy w podrozdziale „Stos wirtualizacji"). Nie jest to jednak prawdą w przypadku maszyn wirtualnych wspieranych przez VA i VSM.

Aby poprawić wydajność w nieklasycznych zagnieżdżonych maszynach wirtualnych i scenariuszach VSM (szczegóły w następnej sekcji), hipernadzorca obsługuje rozszerzenie „oświecające" o nazwie „bezpośrednie wirtualne opróżnianie" (ang. *direct virtual flush*), które dostarcza hipernadzorcy L1 dwa hiperwywołania do bezpośredniego unieważnienia bufora TLB. W szczególności hiperwywołanie *HvFlushGuestPhysicalAddressList* (udokumentowane w specyfikacji funkcjonalnej najwyższego poziomu (TLFS — ang. *Top Level Functional Specification*)) pozwala hipernadzorcy L1 na unieważnienie pojedynczego wpisu w tablicy zagnieżdżonych stron-cieni, usuwając kary wydajnościowe związane z opróżnieniem całej tablicy zagnieżdżonych stron-cieni i wielokrotnymi instrukcjami VMEXIT potrzebnymi do jej odtworzenia.

Eksperyment: włączanie zagnieżdżonej wirtualizacji w platformie Hyper-V

Jak wyjaśniono w tym punkcie, aby uruchomić maszynę wirtualną wewnątrz maszyny wirtualnej L1 w platformie Hyper-V, najpierw włącz funkcję zagnieżdżonej wirtualizacji w systemie hosta. Do tego eksperymentu potrzebna jest stacja robocza z procesorem Intel lub AMD i zainstalowanym systemem Windows 10 lub Windows Server 2019 (wersja minimalna to Aktualizacja rocznicowa RS1). Powinieneś stworzyć maszynę wirtualną typu 2 za pomocą menedżera funkcji Hyper-V (ang. *Hyper-V Manager*) lub Windows PowerShell z co najmniej 4 GB pamięci RAM. W eksperymencie tworzysz zagnieżdżoną maszynę wirtualną L2 w utworzonej maszynie wirtualnej, więc przydziel wystarczającą ilość pamięci.

Po pierwszym uruchomieniu maszyny wirtualnej i wstępnej konfiguracji zamknij maszynę i otwórz administracyjne okno PowerShell (wpisz **`Windows PowerShell`** w polu wyszukiwania na pasku zadań, a następnie kliknij prawym przyciskiem myszy ikonę *PowerShell* i wybierz opcję *Run as administrator* (*Uruchom jako administrator*)). Następnie wpisz następujące polecenie, gdzie ciąg <*NazwaVM*> musisz zastąpić nazwą twojej maszyny wirtualnej:

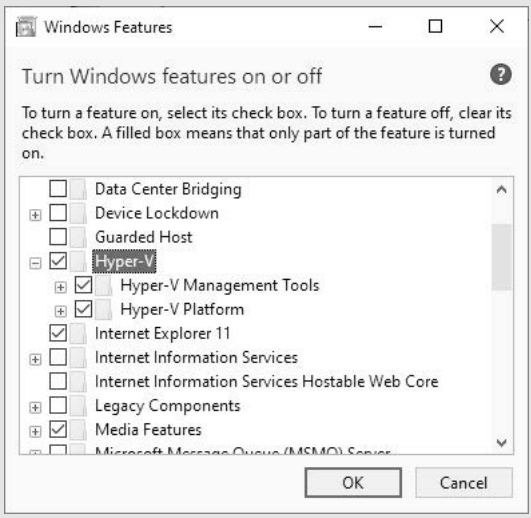
```
Set-VMProcessor -VMName "<NazwaVM>" -ExposeVirtualizationExtensions $true
```

Aby w sposób prawidłowy sprawdzić, czy funkcja wirtualizacji zagnieżdżonej jest poprawnie włączona, polecenie

```
$(Get-VMProcessor -VMName "<NazwaVM>").ExposeVirtualizationExtensions
```

powinno zwrócić wartość `True`.

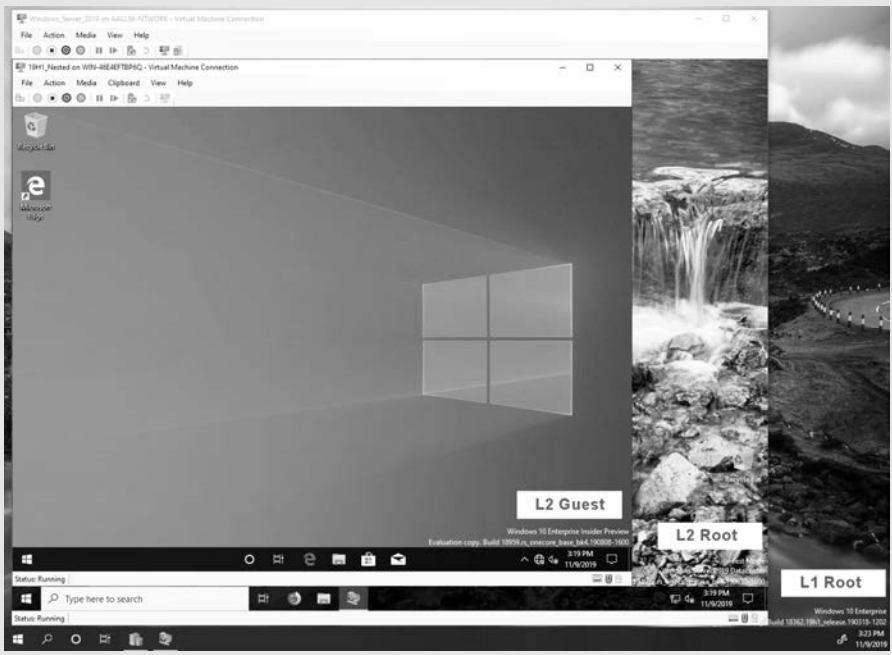
Po włączeniu funkcji wirtualizacji zagnieżdżonej możesz zrestartować swoją maszynę wirtualną. Zanim będziesz mógł uruchomić hipernadzorcę L1 w maszynie wirtualnej, powinieneś dodać niezbędny komponent poprzez Panel sterowania. W maszynie wirtualnej wyszukaj w polu wyszukiwania na pasku zadań *Control Panel* (*Panel sterowania*), otwórz go, kliknij *Programs and Features* (*Programy i funkcje*), a następnie wybierz opcję *Turn Windows features on or off* (*Włącz lub wyłącz funkcje systemu Windows*). Zaznacz całe drzewo Hyper-V, jak pokazano na poniższym rysunku.



Kliknij *OK*. Po zakończeniu procedury kliknij *Restart*, aby ponownie uruchomić maszynę wirtualną (ten krok jest konieczny). Po ponownym uruchomieniu maszyny wirtualnej możesz sprawdzić obecność hipernadzorcy L1 poprzez aplikację *System Information* (*Informacje o systemie*) (wpisz **msinfo32** w polu wyszukiwania na pasku zadań. Aby uzyskać więcej szczegółów, zapoznaj się z eksperymentem „Wykrywanie VBS i jego usług" w dalszej części tego rozdziału). Jeśli hipernadzorca nie został z jakiegoś powodu uruchomiony, możesz go zmusić do uruchomienia, otwierając administracyjny wiersz poleceń w maszynie wirtualnej (wpisz **cmd** w polu wyszukiwania na pasku zadań, a następnie kliknij prawym przyciskiem myszy ikonę *Command Prompt* (*Wiersz polecenia*) i wybierz opcję *Run as administrator* (*Uruchom jako administrator*)) i wprowadzając następujące polecenie:

```
bcdedit /set {current} hypervisorlaunchtype Auto
```

Na tym etapie możesz użyć menedżera funkcji Hyper-V (ang. *Hyper-V Manager*) lub Windows PowerShell, aby utworzyć maszynę wirtualną gościa L2 bezpośrednio w maszynie wirtualnej. W wyniku tego możesz otrzymać coś podobnego do poniższego rysunku:



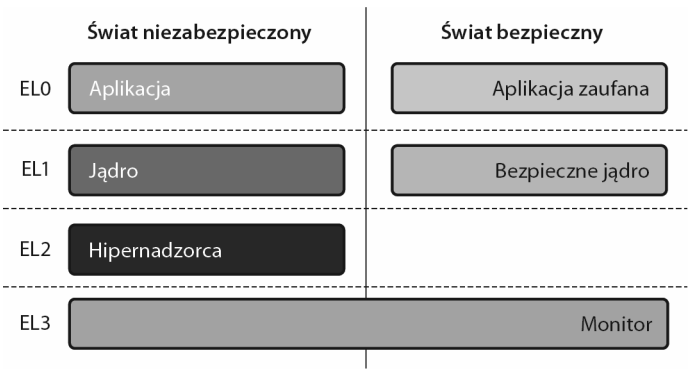
Z partycji głównej (*root*) L2 możesz również włączyć debuger hipernadzorcy L1, w podobny sposób jak wyjaśniono w eksperymencie „Podłączenie debugera hipernadzorcy" we wcześniejszej części tego rozdziału. Jedynym ograniczeniem w chwili pisania tego rozdziału jest to, że nie można używać debugera sieciowego w konfiguracjach zagnieżdżonych; jedyną obsługiwaną konfiguracją przeznaczoną do debugowania hipernadzorcy L1 jest port szeregowy. Oznacza to, że w systemie hosta musisz włączyć dwa wirtualne porty szeregowe w maszynie wirtualnej L1 (jeden dla hipernadzorcy, a drugi dla partycji głównej L2) i dołączyć je do nazwanych potoków. W przypadku maszyn wirtualnych typu 2 użyj następujących poleceń w konsoli PowerShell, aby ustawić dwa porty szeregowe w maszynie wirtualnej L1 (podobnie jak w przypadku poprzednich poleceń, musisz zastąpić ciąg <*NazwaVM*> nazwą swojej maszyny wirtualnej):

```
Set-VMComPort -VMName "<NazwaVM>" -Number 1 -Path \\.\pipe\HV_dbg
Set-VMComPort -VMName "<NazwaVM>" -Number 2 -Path \\.\pipe\NT_dbg
```

Następnie skonfiguruj debuger hipernadzorcy tak, aby był podłączony do portu szeregowego COM1, natomiast debuger jądra NT powinien być podłączony do COM2 (więcej szczegółów w poprzednim eksperymencie).

Hipernadzorca Windows w procesorze ARM64

W przeciwieństwie do architektur x86 i AMD64, w których wsparcie dla wirtualizacji sprzętowej zostało dodane długo po ich pierwotnym zaprojektowaniu, architektura ARM64 została zaprojektowana ze wsparciem dla wirtualizacji sprzętowej. W szczególności, jak pokazano na rysunku 9.22, środowisko wykonawcze ARM64 zostało podzielone na trzy różne domeny bezpieczeństwa (zwane poziomami wyjątków EL (ang. *Exception Levels*)). Poziomy wyjątków określają poziom przywilejów; im wyższy EL, tym więcej uprawnień ma wykonujący się kod. Chociaż wszystkie aplikacje trybu użytkownika działają w EL0, jądro NT (i sterowniki trybu jądra) zwykle działa w EL1. Ogólnie rzecz biorąc, część oprogramowania działa tylko na jednym poziomie wyjątku. EL2 jest poziomem uprawnień przeznaczonym do uruchamiania hipernadzorcy (który w ARM64 jest również nazywany „menedżerem maszyn wirtualnych") i jest wyjątkiem od tej reguły. Hipernadzorca dostarcza usługi wirtualizacji i może działać w świecie bezpiecznym zarówno w EL2, jak i EL1. (EL2 nie istnieje w świecie bezpiecznym. Strefa zaufania procesorów ARM (ang. *TrustZone*) zostanie omówiona w dalszej części tego rozdziału).



RYSUNEK 9.22. *Środowisko wykonawcze procesora ARM64*

W odróżnieniu od architektury AMD64, gdzie procesor wchodzi w tryb *root* (domena wykonawcza, w której działa hipernadzorca) tylko z poziomu kontekstu jądra i przy pewnych założeniach, kiedy standardowe urządzenie ARM64 dokonuje rozruchu (*boot*), oprogramowanie układowe firmware UEFI i menedżer rozruchu rozpoczynają wykonywanie swoich instrukcji w EL2. Na tych urządzeniach program ładujący hipernadzorcę (lub Bezpieczny program uruchamiający (*launcher*), w zależności od przepływu startowego) jest w stanie uruchomić hipernadzorcę bezpośrednio i w późniejszym czasie obniżyć poziom wyjątków do EL1 (emitując instrukcję powrotu wyjątku, znaną również jako ERET).

Nad warstwą poziomów wyjątków technologia strefy zaufania umożliwia podział systemu na dwa stany bezpieczeństwa wykonania: bezpieczny i niezabezpieczony. Bezpieczne oprogramowanie może zasadniczo uzyskać dostęp zarówno do bezpiecznej, jak i niezabezpieczonej pamięci i zasobów, natomiast normalne oprogramowanie może uzyskać dostęp tylko do niezabezpieczonej pamięci i zasobów. Stan niezabezpieczony jest również określany jako *normalny świat*. Pozwala to na równoległe działanie systemu operacyjnego z zaufanym systemem operacyjnym na tym samym sprzęcie i zapewnia ochronę przed niektórymi atakami programowymi i sprzętowymi. W stanie bezpiecznym, określanym również jako *bezpieczny świat*, zwykle uruchamiane są bezpieczne urządzenia (ich oprogramowanie układowe firmware i zakresy IOMMU) i ogólnie wszystko, co wymaga, aby procesor znajdował się w stanie bezpiecznym.

Aby poprawnie komunikować się ze światem bezpiecznym, niezabezpieczony system operacyjny emituje *bezpieczne wywołania metod* (SMC — ang. *secure method calls*), które zapewniają mechanizm podobny do standardowych wywołań systemowych (*syscalls*) systemu operacyjnego. Wywołania SMC są zarządzane przez strefę zaufania. Strefa zaufania zazwyczaj zapewnia separację między światem normalnym i bezpiecznym poprzez cienką warstwę ochrony pamięci, którą zapewniają dobrze zdefiniowane sprzętowe jednostki ochrony pamięci (Qualcomm nazywa je XPU). Jednostki ochrony pamięci XPU są konfigurowane przez oprogramowanie układowe firmware, aby umożliwić dostęp do określonych lokalizacji pamięci tylko określonym środowiskom wykonawczym. (Pamięć świata bezpiecznego nie może być dostępna dla oprogramowania świata normalnego).

W maszynach serwerowych ARM64 Windows jest w stanie bezpośrednio uruchomić hipernadzorcę. Maszyny klienckie często nie mają XPU, nawet jeśli strefa zaufania jest włączona. (Większość urządzeń klienckich ARM64, w których może działać Windows, jest dostarczana przez Qualcomm). W tych urządzeniach klienckich za separację między światem bezpiecznym i normalnym odpowiada zastrzeżony hipernadzorca o nazwie QHEE (ang. *Qualcomm Hypervisor Execution Environment* — środowisko wykonawcze hipernadzorcy firmy Qualcomm), który zapewnia izolację pamięci przy użyciu translacji pamięci fazy drugiej (warstwa ta jest taka sama jak warstwa SLAT używana przez hipernadzorcę Windows). Środowisko QHEE przechwytuje każde bezpieczne wywołanie metod SMC emitowane przez uruchomiony system operacyjny: może przekazać bezpieczne wywołanie metod SMC bezpośrednio do strefy zaufania (po sprawdzeniu niezbędnych praw dostępu) lub wykonać jakąś pracę w jego imieniu. W tych urządzeniach strefa zaufania ponosi również istotną odpowiedzialność za ładowanie i weryfikację autentyczności oprogramowania układowego firmware maszyny oraz koordynuje wspólnie z środowiskiem wykonawczym QHEE prawidłowe wykonanie metody rozruchu (ang. *boot*) typu bezpieczne uruchamianie (ang. *Secure Launch*).

Chociaż w systemie Windows świat bezpieczny zasadniczo nie jest używany (rozróżnienie między światem bezpiecznym a światem niezabezpieczonym jest już zapewniane przez hipernadzorcę poprzez wirtualne poziomy zaufania (VTL)), hipernadzorca platformy Hyper-V nadal działa w EL2. Nie jest to kompatybilne ze środowiskiem wykonawczym hipenadzorcy firmy Qualcomm QHEE, który również działa w EL2. Aby prawidłowo rozwiązać problem, system Windows przyjmuje określoną strategię rozruchu — proces bezpiecznego uruchamiania (*Secure Launch*) jest koordynowany za pomocą QHEE. Kiedy bezpieczne uruchamianie kończy swoją pracę, hipernadzorca QHEE opuszcza pamięć i przekazuje wykonywanie instrukcji hipernadzorcy Windows, który został załadowany do pamięci jako część bezpiecznego uruchamiania. Na późniejszych etapach rozruchu, po uruchomieniu bezpiecznego jądra i utworzeniu przez SMSS pierwszej sesji trybu użytkownika tworzony jest nowy specjalny trustlet (Qualcomm nazwał go „QcExt"). Trustlet działa jak oryginalny hipernadzorca platformy ARM64;

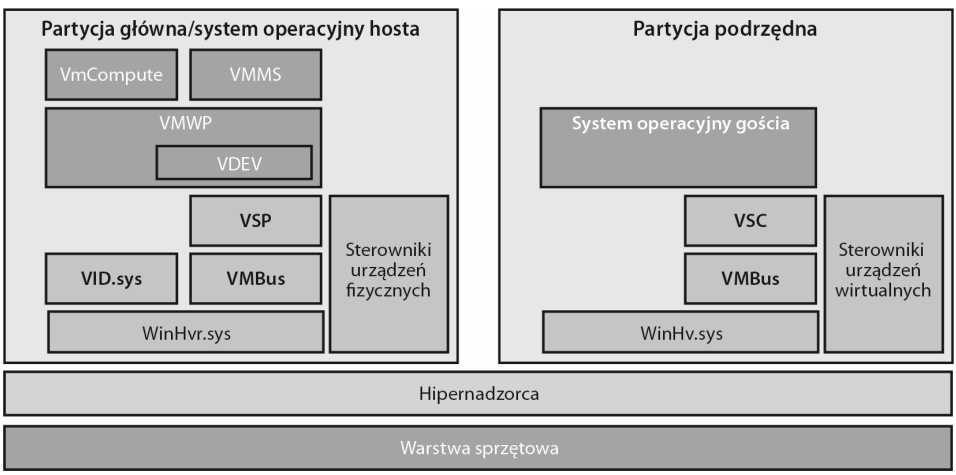
przechwytuje wszystkie żądania SMC, weryfikuje ich integralność, zapewnia potrzebne izolacje pamięci (poprzez usługi prezentowane przez Bezpieczne jądro) i jest w stanie wysyłać i odbierać instrukcje z bezpiecznego monitorowania (ang. *Secure Monitor*) w EL3.

Architektura przechwytywania SMC jest zaimplementowana zarówno w jądrze NT, jak i w trustlecie ARM64 i znajduje się poza zakresem tematyki omawianej w tej książce. Wprowadzenie nowego trustletu pozwoliło większości klienckich maszyn ARM64 na rozruch z domyślnie włączonym bezpiecznym uruchamianiem i wirtualnym trybem bezpiecznym (VSM — ang. *Virtual Secure Mode*). Tryb VSM jest omówiony w dalszej części tego rozdziału.

Stos wirtualizacji

Chociaż hipernadzorca zapewnia izolację i usługi niskiego poziomu, które zarządzają warstwą sprzętową wirtualizacji, cała wysokopoziomowa implementacja maszyn wirtualnych jest zapewniana przez stos wirtualizacji. Stos wirtualizacji zarządza stanami maszyn wirtualnych, zapewnia im pamięć i wirtualizuje warstwę sprzętową, zapewniając wirtualną płytę główną, systemowe oprogramowanie układowe firmware i wiele rodzajów urządzeń wirtualnych (emulowanych, syntetycznych i bezpośredniego dostępu). Stos wirtualizacji zawiera również VMBus, ważny komponent, który zapewnia szybki kanał komunikacyjny pomiędzy maszyną wirtualną gościa a partycją główną i może być dostępny poprzez warstwę abstrakcji biblioteki klienta trybu jądra KMCL (ang. *Kernel Mode Client Library*).

W tym rozdziale omawiamy niektóre ważne usługi świadczone przez stos wirtualizacji i analizujemy jego składniki. Rysunek 9.23 przedstawia główne komponenty stosu wirtualizacji.



RYSUNEK 9.23. *Komponenty stosu wirtualizacji*

Usługa menedżera maszyn wirtualnych i procesy robocze

Usługa menedżera maszyn wirtualnych (*Vmms.exe*) jest odpowiedzialna za udostępnienie interfejsu instrumentacji zarządzania Windows (WMI — ang. *Windows Management Instrumentation*) dla partycji głównej, co umożliwia zarządzanie partycjami podrzędnymi za pomocą wtyczki Konsola zarządzania Microsoft (MMC — ang. *Microsoft Management Console*) lub poprzez konsolę PowerShell.

Usługa zarządzania maszynami wirtualnymi (VMMS — ang. *Virtual Machine Management Service*) zarządza żądaniami otrzymanymi przez interfejs WMI w imieniu maszyny wirtualnej (identyfikowanej wewnętrznie przez GUID), takimi jak start, wyłączenie, zamknięcie, wstrzymanie, wznowienie, ponowne uruchomienie itd. Usługa ta kontroluje ustawienia takie jak to, które urządzenia są widoczne dla partycji podrzędnych oraz jak definiowany jest przydział pamięci i procesora dla każdej partycji. Usługa VMMS zarządza dodawaniem i usuwaniem urządzeń. Kiedy maszyna wirtualna jest uruchamiana, usługa VMMS odgrywa również kluczową rolę w tworzeniu odpowiedniego procesu roboczego maszyny wirtualnej (*VMWP.exe* — ang. *Virtual Machine Worker Process*). Usługa VMMS zarządza migawkami (ang. *snapshots*) maszyn wirtualnych, przekierowując żądania migawki do procesu usługi VMWP, w przypadku gdy maszyna wirtualna jest uruchomiona, lub wykonując samą migawkę w przeciwnym przypadku.

Usługa VMWP wykonuje różne zadania wirtualizacyjne, które wykonywałby typowy monolityczny hipernadzorca (podobnie jak w przypadku programowych rozwiązań wirtualizacyjnych). Oznacza to zarządzanie maszyną stanów dla danej partycji podrzędnej (aby umożliwić obsługę takich funkcji jak migawki i przejścia stanów), odpowiadanie na różne powiadomienia przychodzące z hipernadzorcy, wykonywanie emulacji pewnych urządzeń prezentowanych partycjom podrzędnym (zwanym urządzeniami emulowanymi) oraz współpracę z komponentem obsługi i konfiguracji maszyn wirtualnych. Proces roboczy maszyny wirtualnej pełni ważną rolę polegającą na uruchamianiu wirtualnej płyty głównej oraz utrzymywaniu stanu każdego urządzenia wirtualnego należącego do maszyny wirtualnej. Zawiera również komponenty odpowiedzialne za zdalne zarządzanie stosem wirtualizacji, a także komponent protokołu zdalnego pulpitu (RDP — ang. *Remote Desktop Protocol*), który pozwala za pomocą klienta zdalnego pulpitu połączyć się z dowolną partycją podrzędną i zdalnie wyświetlić jej interfejs użytkownika oraz wejść z nią w interakcję. Proces roboczy maszyny wirtualnej prezentuje obiekty COM, które zapewniają interfejs wykorzystywany przez VMMS (oraz usługę VmCompute), aby komunikować się z instancją usługi VMWP odzwierciedlającą daną maszynę wirtualną.

Usługa obliczeniowa hosta maszyny wirtualnej (zaimplementowana w binariach *Vmcompute.exe* i *Vmcompute.dll*) jest kolejnym ważnym komponentem, który obsługuje większość intensywnych obliczeniowo operacji, które nie są zaimplementowane w usłudze VMMS. Operacje takie jak analiza raportu pamięci maszyny wirtualnej (dla pamięci dynamicznej), zarządzanie plikami VHD i VHDX oraz tworzenie warstw bazowych dla kontenerów są implementowane w usłudze obliczeniowej hosta maszyny wirtualnej. Proces roboczy maszyny wirtualnej i usługa VMMS mogą komunikować się z usługą obliczeniową hosta dzięki obiektom COM, które ta usługa prezentuje.

Usługa menedżera maszyn wirtualnych (VMMS), proces roboczy maszyny wirtualnej (VMWP) oraz usługa obliczeniowa maszyny wirtualnej (VM compute) są w stanie otworzyć i sparsować wiele plików konfiguracyjnych, które prezentują listę wszystkich maszyn wirtualnych utworzonych w systemie oraz konfigurację każdej z nich. W szczególności:

- Repozytorium konfiguracyjne przechowuje listę maszyn wirtualnych zainstalowanych w systemie, ich nazwy, plik konfiguracyjny i identyfikator GUID w pliku *data.vmcx* znajdującym się w folderze *C:\ProgramData\Microsoft\Windows\Hyper-V*.
- Repozytorium magazynu danych maszyny wirtualnej (*VM Data Store*) (część usługi obliczeniowej hosta maszyny wirtualnej — ang. *VM host compute*) jest w stanie otworzyć, odczytać i zapisać plik konfiguracyjny (zwykle z rozszerzeniem *.vmcx*) maszyny wirtualnej, który zawiera listę urządzeń wirtualnych i konfigurację sprzętu wirtualnego.

Repozytorium magazynu danych maszyny wirtualnej służy również do odczytu i zapisu pliku, w którym są zapisywane stany maszyny wirtualnej (plik *VM Save State*). Plik stanów maszyny wirtualnej (*VM State*) jest generowany podczas wstrzymywania pracy maszyny wirtualnej i zawiera zapisany stan uruchomionej maszyny wirtualnej, który może być przywrócony w późniejszym czasie (stan partycji, zawartość pamięci maszyny wirtualnej, stan każdego urządzenia wirtualnego). Pliki konfiguracyjne są formatowane przy użyciu formatu XML składającego się z par klucz/wartość. Zwykłe dane XML są przechowywane w postaci skompresowanej przy użyciu zastrzeżonego formatu binarnego, który dodaje logikę zapisu, aby uczynić go odpornym na awarie zasilania. Dokumentacja formatu binarnego wykracza poza tematykę omawianą w tej książce.

Sterownik VID i menedżer pamięci stosu wirtualizacji

Sterownik wirtualizacji infrastruktury (VID — ang. *Virtual Infrastructure Driver*) *VID.sys* jest prawdopodobnie jednym z najważniejszych składników stosu wirtualizacji. Zapewnia on usługi zarządzania partycjami, pamięcią i procesorem dla maszyn wirtualnych uruchomionych w partycji podrzędnej, udostępniając je procesowi roboczemu maszyny wirtualnej, który rezyduje w głównej partycji (*root*). Proces roboczy maszyny wirtualnej oraz usługi VMMS wykorzystują sterownik VID do komunikacji z hipernadzorcą, dzięki interfejsom zaimplementowanym w sterowniku interfejsu hipernadzorcy Windows (*WinHv.sys* i *WinHvr.sys*), importowanym przez sterownik VID. Interfejsy te zawierają cały kod obsługujący zarządzanie hiperwywołaniami hipernadzorcy i pozwalają systemowi operacyjnemu (lub generycznym sterownikom trybu jądra) na dostęp do hipernadzorcy przy użyciu standardowych wywołań Windows API zamiast hiperwywołań.

Sterownik VID zawiera również menedżera pamięci stosu wirtualizacji. W poprzednim podrozdziale opisaliśmy menedżera pamięci hipernadzorcy, który zarządza fizyczną i wirtualną pamięcią samego hipernadzorcy. Pamięć fizyczna maszyny wirtualnej gościa jest przydzielana i zarządzana przez menedżera pamięci stosu wirtualizacji. Podczas uruchamiania maszyny wirtualnej proces roboczy maszyny wirtualnej (*VMWP.exe*) korzysta z usług menedżera pamięci (zdefiniowanego w interfejsie COM *IMemoryManager*) w celu utworzenia pamięci RAM maszyny wirtualnej gościa. Przydzielanie pamięci maszynie wirtualnej jest procesem dwuetapowym:

1. Proces roboczy maszyny wirtualnej uzyskuje raport o stanie pamięci globalnego systemu (poprzez wykorzystanie usług od balansera pamięci (*Memory Balancer*) w procesie VMMS) i na podstawie dostępnej pamięci systemowej określa rozmiar bloków pamięci fizycznej, które należy zażądać do sterownika VID (poprzez sterowanie we/wy IOCTL *VID_RESERVE*. Rozmiary bloku wahają się od 64 MB do 4 GB). Bloki są przydzielane przez sterownik VID przy pomocy funkcji zarządzania Listą deskryptorów pamięci MDL (ang. *Memory Descriptor List*) (w szczególności *MmAllocatePartitionNodePagesForMdlEx*). Ze względów wydajnościowych i w celu uniknięcia fragmentacji pamięci sterownik VID implementuje algorytm najlepszego starania (ang. *best-effort*) do przydzielania wielkich i dużych stron fizycznych (1 GB i 2 MB), zanim będzie polegać na standardowych małych stronach. Po przydzieleniu bloków pamięci ich strony są deponowane do wewnętrznego wiadra „rezerwowego" utrzymywanego przez sterownik VID. Wiadro zawiera listy stron uporządkowane w tablicy na podstawie parametru stron pamięci takiego jak jakość usługi QOS (ang. *Quality of service*). Parametr QOS jest określany na podstawie typu strony (*huge*, *large*, *small*) i węzła NUMA, do którego należą strony pamięci. Proces ten w nomenklaturze VID nazywany jest „rezerwowaniem pamięci fizycznej" (nie należy go mylić z terminem „rezerwowanie pamięci wirtualnej", koncepcją menedżera pamięci NT).

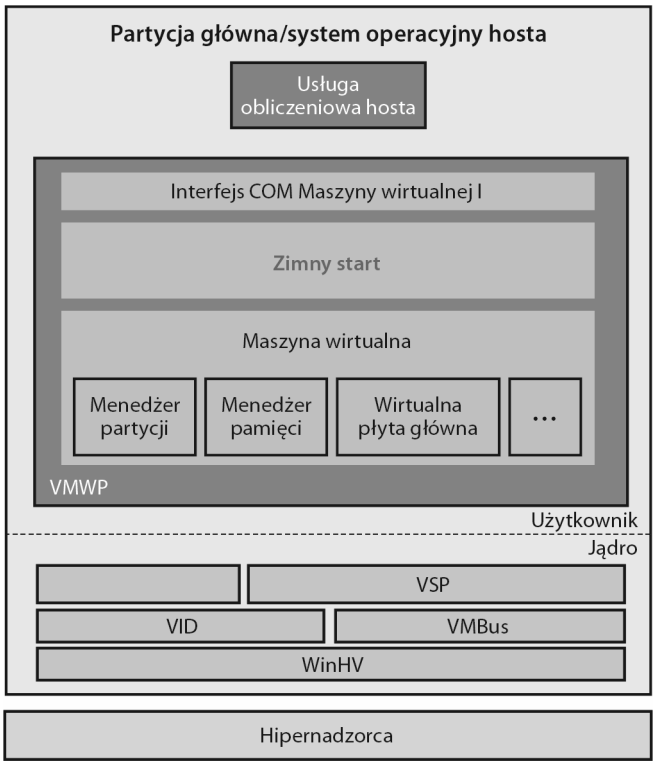
2. Z perspektywy stosu wirtualizacji, *zadeklarowanie* pamięci fizycznej jest procesem opróżniania zarezerwowanych stron w wiadrze i przenoszenia ich w bloku pamięci VID (struktura danych *VSMM_MEMORY_BLOCK*), który jest tworzony i posiadany przez proces roboczy maszyny wirtualnej przy użyciu usług sterownika VID. W procesie tworzenia bloku pamięci sterownik VID najpierw deponuje dodatkowe strony fizyczne w hipernadzorcy (poprzez sterownik Winhvr i hiperwywołanie *HvDepositMemory*). Dodatkowe strony są potrzebne do utworzenia hierarchii stron tablicy SLAT maszyny wirtualnej. Następnie sterownik VID żąda od hipernadzorcy mapowania stron fizycznych opisujących całą pamięć RAM partycji gościa. Hipernadzorca wstawia prawidłowe wpisy do tablicy SLAT i ustawia ich odpowiednie uprawnienia. Tworzona jest fizyczna przestrzeń adresowa partycji gościa. Zakres GPA jest wstawiany do listy należącej do partycji VID. Blok pamięci VID jest własnością procesu roboczego maszyny wirtualnej. Jest on również wykorzystywany do śledzenia pamięci gościa oraz w blokach pamięci wspieranych przez plik DAX. (Więcej szczegółów na temat woluminów DAX i PMEM znajduje się w rozdziale 11., „Buforowanie i systemy plików"). Proces roboczy maszyny wirtualnej może później wykorzystać blok pamięci do wielu celów — na przykład do uzyskania dostępu do niektórych stron podczas zarządzania urządzeniami emulowanymi.

Narodziny maszyny wirtualnej (VM)

Proces uruchamiania maszyny wirtualnej jest zarządzany przede wszystkim przez proces VMMS i VMWP. Kiedy żądanie uruchomienia maszyny wirtualnej (wewnętrznie identyfikowanej przez GUID) jest dostarczane do usługi VMMS (poprzez PowerShell lub aplikację GUI menedżer funkcji Hyper-V (ang. *Hyper-V Manager*)), usługa VMMS rozpoczyna proces uruchamiania od odczytania konfiguracji maszyny wirtualnej z repozytorium magazynu danych, które zawiera GUID maszyny wirtualnej oraz listę wszystkich urządzeń wirtualnych (ang. *Virtual devices VDEV*) składających się na jej wirtualny sprzęt. Następnie weryfikuje, czy ścieżka zawierająca dysk VHD (lub VHDX) odzwierciedlający wirtualny dysk twardy maszyny wirtualnej posiada poprawną listę kontroli dostępu ACL (więcej szczegółów w dalszej części). W przypadku gdy lista ACL nie jest poprawna, o ile jest to określone przez konfigurację maszyny wirtualnej, usługa VMMS (działająca pod kontrolą konta SYSTEM) zapisuje nową listę ACL, zgodną z nowym wystąpieniem procesu VMWP. VMMS wykorzystuje usługi COM do komunikacji z Usługą obliczeniową hosta (HCS — ang. *Host Compute Service*) w celu utworzenia nowego wystąpienia procesu VMWP.

HCS uzyskuje ścieżkę procesu roboczego maszyny wirtualnej poprzez zapytanie o dane rejestracyjne COM znajdujące się w rejestrze Windows ((klucz `HKCU\CLSID\{f33463e0-7d59-`
`↪11d9-9916-0008744f51f3}`). Następnie tworzy nowy proces przy użyciu dobrze zdefiniowanego tokena dostępu, który jest budowany z wykorzystaniem SID maszyny wirtualnej w roli właściciela. W rzeczywistości uprawnienia NT modelu bezpieczeństwa Windows definiuje dobrze znaną wartość sub-organu autoryzującego (83) w celu identyfikacji maszyn wirtualnych (więcej informacji na temat składników bezpieczeństwa systemu można znaleźć w części I, w rozdziale 7., „Bezpieczeństwo"). HCS czeka na zakończenie inicjalizacji procesu VMWP (w ten sposób odsłonięte interfejsy COM stają się gotowe). Wykonywanie instrukcji wraca do usługi VMMS, która może ostatecznie zażądać uruchomienia maszyny wirtualnej do procesu VMWP (poprzez odsłonięty interfejs COM maszyny wirtualnej *IVirtualMachine*).

Jak pokazano na rysunku 9.24, proces roboczy maszyny wirtualnej wykonuje transformację stanu „zimnego startu" dla maszyny wirtualnej. W procesie roboczym maszyny wirtualnej cała maszyna wirtualna jest zarządzana poprzez usługi udostępniane przez „wirtualną płytę główną". Wirtualna płyta główna emuluje płytę główną Intel i440BX w maszynach wirtualnych pierwszej generacji, natomiast w maszynach drugiej generacji emuluje płytę główną własnej konstrukcji. Zarządza ona i utrzymuje listę urządzeń wirtualnych oraz wykonuje transformacje stanów dla każdego z nich. Jak zostało to opisane w następnym rozdziale, każde urządzenie wirtualne jest zaimplementowane jako obiekt COM (prezentujący interfejs *VirtualDevice*) w pliku DLL. Wirtualna płyta główna wylicza każde urządzenie wirtualne z konfiguracji maszyny wirtualnej i ładuje odpowiedni obiekt COM odzwierciedlający dane urządzenie.



RYSUNEK 9.24. *Proces roboczy maszyny wirtualnej i jego interfejs umożliwiający wykonanie „zimnego startu" maszyny wirtualnej*

Proces roboczy maszyny wirtualnej rozpoczyna procedurę uruchamiania od zarezerwowania zasobów potrzebnych każdemu urządzeniu wirtualnemu. Następnie konstruuje fizyczną przestrzeń adresową maszyny wirtualnej gościa (wirtualną pamięć RAM), przydzielając pamięć fizyczną z partycji głównej poprzez sterownik VID. Na tym etapie może uruchomić zasilanie wirtualnej płyty głównej, która będzie się cyklicznie przełączać pomiędzy poszczególnymi urządzeniami wirtualnymi VDEV i je zasilać. Procedura włączania zasilania jest inna dla każdego urządzenia: na przykład urządzenia syntetyczne zwykle komunikują się z własnym dostawcą usług wirtualizacji VSP w celu przeprowadzenia wstępnej konfiguracji.

Jednym z urządzeń wirtualnych, które zasługuje na głębsze omówienie, jest wirtualny BIOS (zaimplementowany w bibliotece *Vmchipset.dll*). Jego metoda włączania zasilania pozwala na włączenie do maszyny wirtualnej początkowego oprogramowania układowego firmware wykonywanego w momencie uruchomienia procesora wirtualnego skryptu programowego bootstrap. VDEV BIOS-u pobiera z sekcji zasobów własnej biblioteki wspierającej odpowiednie dla danej maszyny wirtualnej oprogramowanie układowe firmware (tzw. legacy BIOS w przypadku maszyn wirtualnych pierwszej generacji; UEFI w pozostałych przypadkach), buduje część oprogramowania układowego firmware zawierającego konfigurację lotną (jak tablica ACPI i SRAT) i wstrzykuje ją do odpowiedniej pamięci fizycznej gościa, korzystając z usług świadczonych przez sterownik VID. Sterownik VID rzeczywiście jest w stanie mapować zakresy pamięci opisane przez blok pamięci VID w pamięci trybu użytkownika, dostępnej dla procesu roboczego maszyny wirtualnej (procedura ta jest wewnętrznie nazywana „tworzeniem apertury pamięci").

Po pomyślnym zasileniu wszystkich urządzeń wirtualnych Proces roboczy maszyny wirtualnej może uruchomić wirtualny procesor bootstrapowy maszyny wirtualnej, wysyłając odpowiedni IOCTL do sterownika VID, który uruchomi procesor wirtualny i jego pompę komunikatów (używaną do wymiany komunikatów pomiędzy sterownikiem VID a procesem roboczym maszyny wirtualnej).

Eksperyment: zrozumienie bezpieczeństwa procesu roboczego maszyny wirtualnej oraz plików wirtualnego dysku twardego

W poprzedniej sekcji omówiliśmy, w jaki sposób proces roboczy maszyny wirtualnej jest uruchamiany przez usługę obliczeniową hosta (*Vmcompute.exe*), gdy żądanie uruchomienia maszyny wirtualnej jest dostarczane do procesu VMMS (poprzez WMI). Przed skomunikowaniem się z HCS, VMMS generuje token bezpieczeństwa dla nowego wystąpienia procesu roboczego maszyny wirtualnej.

Do modelu zabezpieczeń systemu Windows dodano trzy nowe encje, aby odpowiednio obsłużyć maszyny wirtualne (model zabezpieczeń systemu Windows został obszernie omówiony w rozdziale 7. części I):

- Grupa zabezpieczeń „maszyny wirtualne", identyfikowana za pomocą identyfikatora zabezpieczeń S-1-5-83-0.
- Identyfikator bezpieczeństwa maszyny wirtualnej SID, oparty na unikalnym identyfikatorze GUID maszyny wirtualnej. Identyfikator SID maszyny wirtualnej staje się właścicielem tokena bezpieczeństwa wygenerowanego dla procesu roboczego maszyny wirtualnej.
- Zdolność bezpieczeństwa procesu roboczego maszyny wirtualnej używana do zapewnienia aplikacjom działającym w kontenerach aplikacji (*AppContainers*) dostępu do usług platformy Hyper-V wymaganych przez proces roboczy maszyny wirtualnej.

W tym eksperymencie utworzysz nową maszynę wirtualną za pośrednictwem menedżera Hyper-V w lokalizacji dostępnej tylko dla aktualnego użytkownika i grupy administratorów, a następnie sprawdzisz, jak zmieniają się zabezpieczenia plików maszyny wirtualnej i procesu roboczego maszyny wirtualnej.

W pierwszej kolejności otwórz administracyjny wiersz poleceń i utwórz folder w jednym z woluminów stacji roboczej (w przykładzie użyliśmy *C:\TestVm*), używając następującego polecenia:

```
md c:\TestVm
```

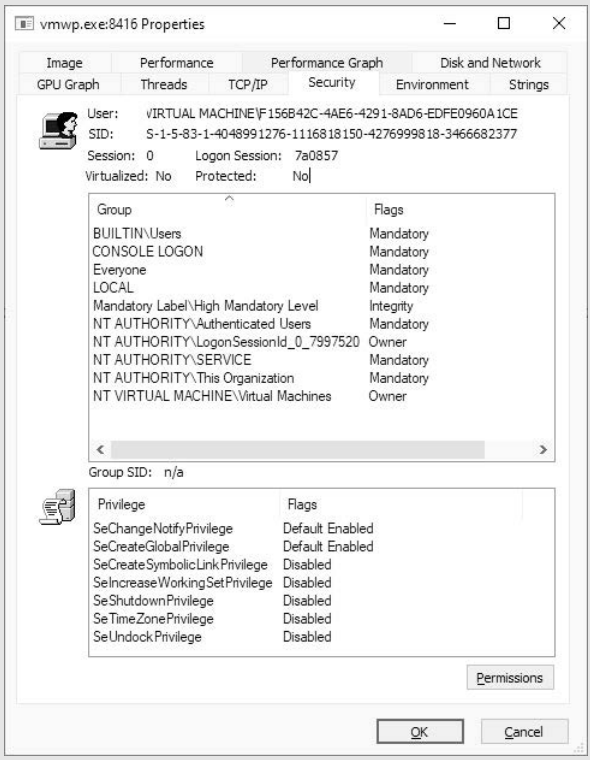
Następnie usuń wszystkie odziedziczone wpisy ACE (ang. *Access Control Entries*); więcej szczegółów w rozdziale 7. części I) i dodaj wpisy pełnego dostępu ACE dla grupy administratorów i aktualnie zalogowanego użytkownika. Poniższe polecenia wykonują opisane czynności (zastąp `C:\TestVm` ścieżką do swojego katalogu, a *<Nazwa>* nazwą aktualnie zalogowanego użytkownika):

```
icacls c:\TestVm /inheritance:r
icacls c:\TestVm /grant Administrators:(CI)(OI)F
icacls c:\TestVm /grant <Nazwa>:(CI)(OI)F
```

Aby sprawdzić, czy folder ma prawidłową listę ACL, otwórz Eksploratora plików (naciskając na klawiaturze klawisze *Win+E*), kliknij prawym przyciskiem myszy folder, wybierz *Properties* (*Właściwości*), a na koniec kliknij zakładkę *Security* (*Zabezpieczenia*). Powinno pojawić się okno podobne do poniższego:

Otwórz menedżera funkcji Hyper-V (ang. *Hyper-V Manager*), utwórz maszynę wirtualną (i jej odpowiedni dysk wirtualny), a następnie zapisz ją w nowo utworzonym folderze (procedura dostępna na stronie: *https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/create-virtual-machine*). W przypadku tego eksperymentu nie ma potrzeby instalowania systemu operacyjnego na maszynie wirtualnej. Po zakończeniu działania kreatora nowej maszyny wirtualnej uruchom maszynę wirtualną (w tym przykładzie jest to VM1).

Otwórz eksplorator procesów (Process Explorer) jako administrator i zlokalizuj proces *vmwp.exe*. Kliknij go prawym przyciskiem myszy i wybierz *Properties* (właściwości). Zgodnie z oczekiwaniami widać, że procesem nadrzędnym jest *vmcompute.exe* (usługa obliczeniowa hosta). Jeśli klikniesz zakładkę *Security* (zabezpieczenia), powinieneś zobaczyć, że SID maszyny wirtualnej jest ustawiony jako właściciel procesu, a token należy do grupy *Virtual Machines* (maszyny wirtualne):



SID jest skomponowany poprzez odzwierciedlenie GUID maszyny wirtualnej. W tym przykładzie GUID maszyny wirtualnej to *{F156B42C-4AE6-4291-8AD6-EDFE0960A1CE}*. (Możesz to sprawdzić również za pomocą konsoli PowerShell, jak wyjaśniono to w eksperymencie „Praktyczne korzystanie z głównego planisty" we wcześniejszej części tego rozdziału). GUID jest sekwencją 16-bajtów, zorganizowaną jako jedna 32-bitowa (4-bajtowa) liczba całkowita, dwie 16-bitowe (2-bajtowe) liczby całkowite i 8 bajtów końcowych. GUID w tym przykładzie jest zorganizowany jako:

- 0xF156B42C jako pierwsza 32-bitowa liczba całkowita, która w systemie dziesiętnym wynosi 4048991276.
- 0x4AE6 i 0x4291 jako dwie 16-bitowe liczby całkowite, które połączone jako jedna 32-bitowa wartość to 0x42914AE6, czyli 1116818150 w systemie dziesiętnym (pamiętaj, że system jest typu *little endian*, więc mniej znaczący bajt znajduje się pod niższym adresem).
- Ostatnia sekwencja bajtów to 0x8A, 0xD6, 0xED, 0xFE, 0x09, 0x60, 0xA1 i 0xCE (trzecia część przedstawionego czytelnego dla człowieka identyfikatora GUID, 8AD6, jest sekwencją bajtów, a nie wartością 16-bitową), która połączona jako dwie wartości 32-bitowe to 0xFEEDD68A i 0xCEA16009, czyli 4276999818 i 3466682377 w systemie dziesiętnym.

Jeśli połączymy wszystkie wyliczone liczby dziesiętne z ogólnym identyfikatorem SID emitowanym przez organ autoryzujący NT (S-1-5) i bazowym RID maszyny wirtualnej (83), powinniśmy otrzymać ten sam SID pokazany w eksploratorze procesów (na przykładzie S-1-5-83-4048991276-1116818150-4276999818-3466682377).

Jak widać z eksploratora procesów, token bezpieczeństwa procesu VMWP nie zawiera grupy *Administrators* (*Administratorzy*) i nie został utworzony w imieniu zalogowanego użytkownika. Jak to możliwe, że proces roboczy maszyny wirtualnej ma dostęp do wirtualnego dysku twardego i plików konfiguracyjnych maszyny wirtualnej?

Odpowiedź leży w procesie VMMS, który podczas tworzenia maszyny wirtualnej skanuje każdy element ścieżki dostępu do maszyny i modyfikuje uznaniową listę kontroli dostępu (DACL — ang. *discretionary access control list*) potrzebnych folderów i plików. W szczególności folder główny maszyny wirtualnej (folder główny ma taką samą nazwę jak maszyna wirtualna, więc w utworzonym katalogu powinieneś znaleźć podfolder o takiej samej nazwie jak Twoja maszyna wirtualna) jest dostępny dzięki dodanej grupie bezpieczeństwa wpisów kontroli dostępu (ACE — ang. *Access Control Entry*) do maszyn wirtualnych. Plik wirtualnego dysku twardego jest natomiast dostępny dzięki wpisom ACE typu zawsze-dozwolony, wskazującym na SID maszyny wirtualnej.

Można to sprawdzić za pomocą Eksploratora plików: otwórz folder wirtualnego dysku twardego maszyny wirtualnej (folder o nazwie *Virtual Hard Disks*, znajdujący się w folderze głównym maszyny wirtualnej), kliknij prawym przyciskiem myszy plik VHDX (lub VHD), wybierz opcję *Properties*, a następnie kliknij stronę *Security*. Powinieneś zobaczyć dwa nowe ACE inne niż ustawione początkowo. (Jeden to wpis ACE maszyny wirtualnej; drugi to zdolność obsługi przez proces roboczy maszyny wirtualnej kontenerów aplikacji).

Jeśli zatrzymasz maszynę wirtualną i spróbujesz usunąć wpis ACE maszyny wirtualnej z pliku, zobaczysz, że maszyna wirtualna nie potrafi się już uruchomić. Aby przywrócić prawidłowy ACL dla wirtualnego dysku twardego, możesz uruchomić skrypt PowerShell dostępny pod adresem *https://gallery.technet.microsoft.com/Hyper-V-Restore-ACL-e64dee58*.

VMBus

VMBus jest mechanizmem prezentowanym przez stos wirtualizacji platformy Hyper-V w celu zapewnienia wewnętrznej komunikacji między partycjami pomiędzy maszynami wirtualnymi. Jest to wirtualna magistrala, która tworzy kanały pomiędzy gościem a gospodarzem. Kanały te umożliwiają współdzielenie danych pomiędzy partycjami i konfigurowanie urządzeń parawirtualizowanych (zwanych też syntetycznymi).

W partycji głównej (*root*) znajdują się dostawcy usług wirtualizacji VSP, którzy komunikują się przez magistralę VMBus w celu obsługi żądań urządzeń z partycji podrzędnych. Z drugiej strony partycje podrzędne (lub partycje gości) używają konsumentów usługi wirtualizacji (VSC — ang. *Virtualization Service Consumer*) do przekierowania żądań urządzeń do VSP poprzez VMBus. Partycje podrzędne wymagają sterowników VMBus i VSC do korzystania z parawirtualizowanych stosów urządzeń (więcej szczegółów na temat obsługi sprzętu wirtualnego znajduje się w dalszej części tego rozdziału, w punkcie „Obsługa sprzętu wirtualnego"). Kanały magistrali maszyny wirtualnej VMBus pozwalają klientowi usługi wirtualizacji (VSC) oraz dostawcy usługi wirtualizacji (VSP) na przesyłanie danych przede wszystkim przez dwa bufory pierścieniowe: bufor wyższego szczebla (*upstream*) i bufor niższego szczebla (*downstream*). Te bufory pierścieniowe są mapowane do obu partycji dzięki hipernadzorcy, który — jak omówiono w poprzednim punkcie — dostarcza również usługi komunikacji między partycjami poprzez SynIC.

Jednym z pierwszych urządzeń wirtualnych VDEV, które proces roboczy maszyny wirtualnej uruchamia podczas włączania maszyny wirtualnej, jest VMBus VDEV (zaimplementowany w *Vmbusvdev.dll*). Jego program włączania łączy proces roboczy maszyny wirtualnej ze sterownikiem głównym VMBus (*Vmbusr.sys*), wysyłając instrukcję sterowania we/wy (IOCTL) `VMBUS_VDEV_SETUP` do urządzenia głównego VMBus (o nazwie `\Device\RootVmBus`). Sterownik główny VMBus koordynuje nadrzędny punkt końcowy dwukierunkowej komunikacji z maszyną wirtualną podrzędną. Jego początkowy program konfiguracyjny, który jest wywoływany w momencie, gdy docelowa maszyna wirtualna nie jest jeszcze włączona, ma za zadanie utworzenie struktury danych XPartition. Struktura ta jest używana do odwzorowywania wystąpienia VMBus dla podrzędnej maszyny wirtualnej oraz do podłączenia potrzebnych syntetycznych źródeł przerwań SynIC (znanych również jako SINT. Więcej szczegółów znajduje się w punkcie „Syntetyczny kontroler przerwań (SynIC)" we wcześniejszej części tego rozdziału). W partycji głównej, VMBus używa dwóch źródeł przerwań syntetycznych: jednego dla wstępnej wymiany komunikatów (co ma miejsce przed utworzeniem kanału) i drugiego dla zdarzeń syntetycznych sygnalizowanych przez bufory pierścieniowe. Partycje podrzędne używają jednak tylko jednego syntetycznego źródła przerwań SINT. Program konfiguracyjny przydziela główny port komunikatów w maszynie wirtualnej podrzędnej oraz odpowiadające mu połączenie w partycji głównej (*root*), a także dla każdego procesora wirtualnego należącego do maszyny wirtualnej przydziela port zdarzeń i jego połączenie (używane do odbierania zdarzeń syntetycznych z maszyny wirtualnej podrzędnej).

Dwa syntetyczne źródła przerwań są mapowane za pomocą dwóch programów ISR, nazwanych *KiVmbusInterrupt0* i *KiVmbusInterrupt1*. Dzięki tym dwóm programom partycja główna jest gotowa do odbierania syntetycznych przerwań i wiadomości od podrzędnej maszyny wirtualnej. Po odebraniu komunikatu (lub zdarzenia) program ISR ustawia w kolejce Wywołanie odroczonej procedury DPC, które sprawdza, czy komunikat jest ważny; jeśli tak, to ustawia w kolejce element roboczy, który zostanie przetworzony później przez system działający na pasywnym poziomie IRQL (co ma dalsze konsekwencje dla kolejki komunikatów).

Gdy VMBus w partycji głównej jest już gotowy, każdy sterownik VSP w głównej partycji może skorzystać z usług udostępnianych przez bibliotekę klienta trybu jądra VMBus, aby przydzielić i zaoferować kanał VMBus maszynie wirtualnej podrzędnej. Biblioteka klienta trybu jądra VMBus (KMCL) odzwierciedla kanał VMBus za pomocą nieprzejrzystej struktury danych *KMODE_CLIENT_CONTEXT*, która jest przydzielana i inicjalizowana w czasie tworzenia kanału (w czasie gdy VSP wywołuje interfejs API *VmbChannelAllocate*). Następnie główny VSP normalnie oferuje kanał podrzędnej maszynie wirtualnej poprzez wywołanie API *VmbChannelEnabled* (ta funkcja w maszynie podrzędnej ustanawia rzeczywiste połączenie z partycją główną (*root*) poprzez otwarcie kanału). KMCL jest implementowany w dwóch sterownikach: jednym działającym w partycji głównej (*root*) (*Vmbkmclr.sys*) i jednym ładowanym w partycjach podrzędnych (*Vmbkmcl.sys*).

Oferowanie kanału w partycji głównej (*root*) jest stosunkowo skomplikowaną operacją, która obejmuje następujące kroki:

1. Sterownik KMCL komunikuje się ze sterownikiem głównym VMBus poprzez obiekt plikowy zainicjalizowany w programie włączania zasilania urządzeń wirtualnych VDEV. Sterownik VMBus uzyskuje strukturę danych XPartition odzwierciedlającą partycję podrzędną i rozpoczyna proces oferowania kanału.
2. Usługi niższego poziomu dostarczane przez sterownik VMBus przydzielają i inicjalizują strukturę danych *LOCAL_OFFER* odzwierciedlającą pojedynczą „ofertę kanału” i wstępnie przydzielają niektóre predefiniowane komunikaty SynIC. VMBus tworzy następnie syntetyczny port zdarzeń w partycji głównej (*root*), z którego maszyna podrzędna może połączyć się ze zdarzeniami sygnałowymi po zapisaniu danych do bufora pierścieniowego. Struktura danych *LOCAL_OFFER* odzwierciedlająca oferowany kanał jest dodawana do wewnętrznej listy kanałów serwera.
3. Po utworzeniu kanału VMBus próbuje wysłać do maszyny podrzędnej wiadomość *OfferChannel*, której celem jest poinformowanie go o nowym kanale. Jednak na tym etapie działanie magistrali VMBus kończy się niepowodzeniem, ponieważ drugi koniec (maszyna wirtualna podrzędna) nie jest jeszcze gotowy i nie rozpoczął wstępnej wymiany (ang. *handshake*) komunikatów.

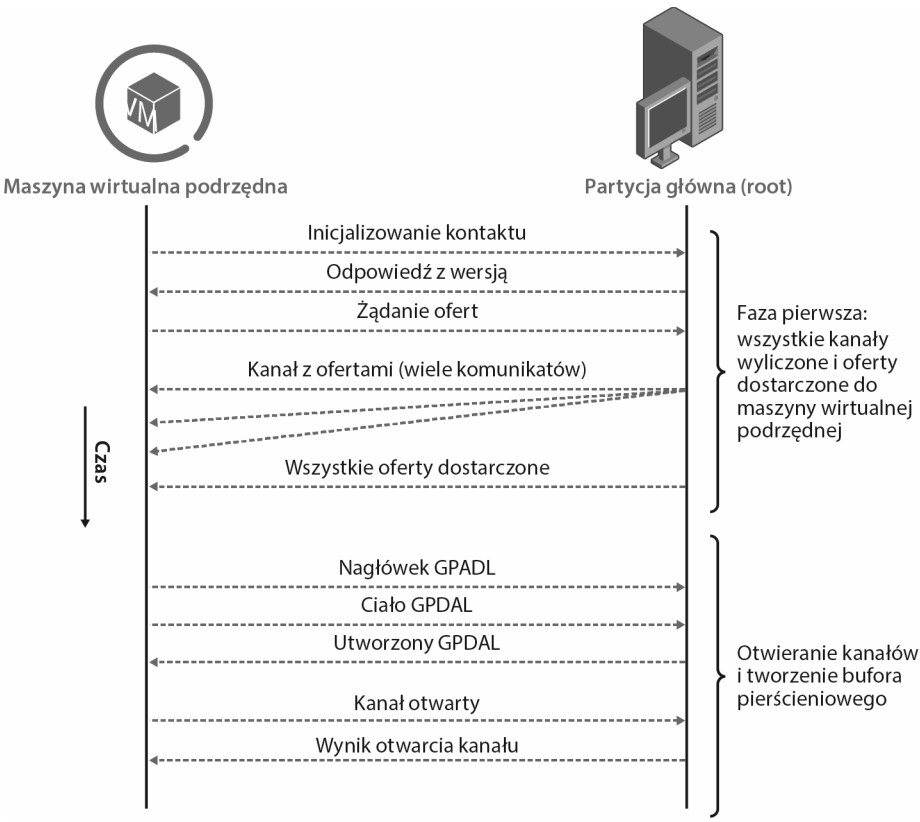
Po zakończeniu oferowania kanałów przez wszystkich dostawców usług wirtualizacji i włączeniu zasilania wszystkich urządzeń wirtualnych VDEV (szczegółowe informacje znajdują się w poprzedniej sekcji) proces roboczy maszyny wirtualnej uruchamia maszynę wirtualną. Aby kanały zostały w pełni zainicjalizowane, a ich odpowiednie połączenia uruchomione, partycja gościa powinna załadować i uruchomić sterownik podrzędny VMBus (*Vmbus.sys*).

Wstępna wymiana komunikatów przez VMBus

W systemie Windows sterownik podrzędny magistrali VMBus jest sterownikiem magistrali WDF wyliczanym i uruchamianym przez menedżera Pnp i znajdującym się w głównym module wyliczania (ang. *enumerator*) ACPI. (Inna wersja sterownika podrzędnego magistrali VMBus jest również dostępna dla systemu Linux. VMBus dla Linuksa nie jest jednak przedmiotem niniejszej książki). Gdy jądro NT startuje w podrzędnej maszynie wirtualnej, sterownik VMBus rozpoczyna swoje wykonanie od inicjalizacji własnego stanu wewnętrznego (co oznacza przydzielenie potrzebnej struktury danych i elementów roboczych) oraz od utworzenia \Device\VmBus głównego (*root*) obiektu urządzenia funkcjonalnego FDO. Następnie menedżer Pnp wywołuje program obsługi przydziału zasobów VMBus. Ta ostatnia konfiguruje właściwe źródło SINT (emitując hiperwywołanie *HvSetVpRegisters* na jednym z rejestrów *HvRegisterSint*, za pomocą sterownika WinHv) i łączy je z programem ISR *KiVmbusInterrupt2*. Ponadto uzyskuje on

stronę SIMP, używaną do wysyłania i odbierania syntetycznych komunikatów do i z partycji głównej (więcej szczegółów w punkcie „Syntetyczny kontroler przerwań (SynIC)" we wcześniejszej części tego rozdziału), oraz tworzy strukturę danych XPartition odzwierciedlającą partycję macierzystą (główną).

Kiedy żądanie uruchomienia obiektu urządzenia funkcjonalnego FDO magistrali VMBus przychodzi od menedżera Pnp, sterownik VMBus rozpoczyna wstępne przekazywanie wiadomości. Na tym etapie każdy komunikat jest wysyłany poprzez emisję hiperwywołania *HvPostMessage* (za pomocą sterownika WinHv), który pozwala hipernadzorcy wstrzyknąć syntetyczne przerwanie do partycji docelowej (w tym przypadku celem jest partycja). Odbiornik pozyskuje wiadomość, po prostu czytając ze strony SIMP; odbiornik sygnalizuje, że wiadomość została odczytana z kolejki poprzez ustawienie nowego typu wiadomości na *MessageTypeNone* (sprawdź specyfikację TLFS, aby dowiedzieć się więcej). Czytelnik powinien postrzegać wstępną wymianę komunikatów (ang. *handshake*), która jest przedstawiona na rysunku 9.25, jako proces podzielony na dwie fazy.



RYSUNEK 9.25. *Schemat wstępnej wymiany komunikatów (ang. handshake) poprzez magistralę VMBus*

Pierwsza faza jest odzwierciedlana przez wiadomość *Initiate Contact*, która jest dostarczana jeden raz w całym czasie życia maszyny wirtualnej. Wiadomość ta jest wysyłana z podrzędnej maszyny wirtualnej do partycji głównej (*root*) w celu wynegocjowania wersji protokołu VMBus obsługiwanej przez obie strony. W chwili pisania tego tekstu istnieje pięć głównych wersji protokołu VMBus, z kilkoma dodatkowymi niewielkimi odmianami. Partycja *root* parsuje

wiadomość, prosi hipernadzorcę o zmapowanie stron monitorujących przydzielonych przez klienta (jeśli protokół to umożliwia) i zwraca odpowiedź, akceptując proponowaną wersję protokołu. W przeciwnym razie (co zdarza się, gdy wersja systemu Windows uruchomiona na partycji głównej jest niższa niż wersja uruchomiona w podrzędnej maszynie wirtualnej), podrzędna maszyna wirtualna ponownie uruchamia proces, obniżając wersję protokołu VMBus, aż do momentu ustalenia wersji zgodnej. W tym momencie maszyna podrzędna jest gotowa do wysłania wiadomości *Request Offers*, która powoduje, że partycja główna wysyła listę wszystkich kanałów już oferowanych przez VSP. Dzięki temu partycja podrzędna może otworzyć kanały później w protokole wymiany komunikatów (ang. *handshake*).

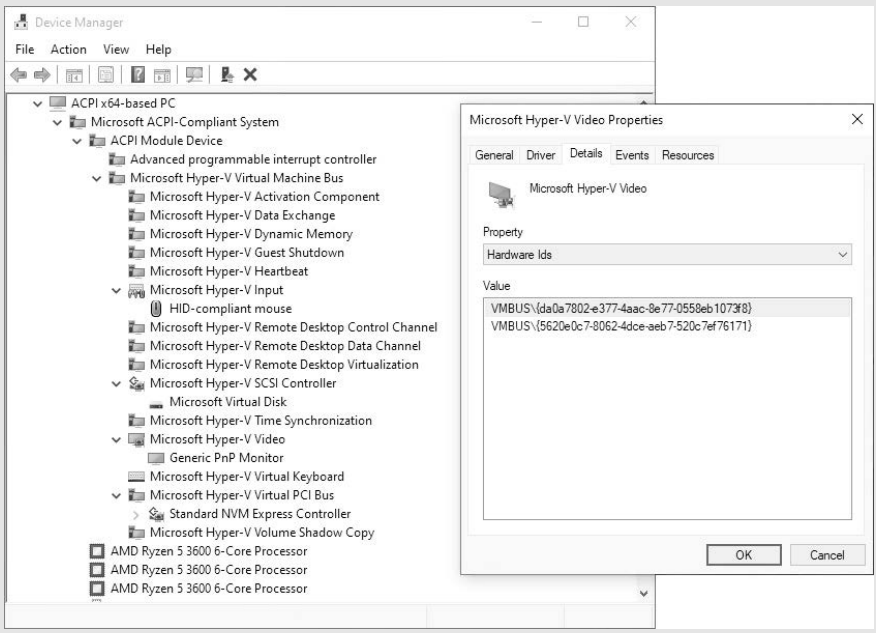
Rysunek 9.25 podkreśla różne syntetyczne komunikaty dostarczane przez hipernadzorcę w celu skonfigurowania kanału lub kanałów VMBus. Partycja główna (*root*) przegląda listę oferowanych kanałów znajdujących się na liście kanałów serwera (struktura danych *LOCAL_OFFER*, omówiona wcześniej) i dla każdego z nich wysyła do podrzędnej maszyny wirtualnej komunikat *Offer Channel*. Wiadomość ta jest taka sama jak ta wysyłana w końcowej fazie protokołu oferowania kanałów, którą omówiliśmy wcześniej w punkcie „VMBus". Tak więc, o ile pierwsza faza wstępnej wymiany komunikatów (ang. *handshake*) ma miejsce tylko jeden raz w całym czasie życia maszyny wirtualnej, o tyle druga faza może rozpocząć się w dowolnym momencie, gdy kanał zostanie zaoferowany. Wiadomość *Offer Channel* zawiera ważne dane służące do jednoznacznej identyfikacji kanału, takie jak typ kanału i identyfikator GUID wystąpienia. W przypadku kanałów VDEV, te dwa GUID są wykorzystywane przez menedżera Pnp do prawidłowego zidentyfikowania powiązanego urządzenia wirtualnego.

Maszyna podrzędna odpowiada na wiadomość, przydzielając strukturę danych klienta *LOCAL_OFFER* odzwierciedlającą kanał i odpowiedni obiekt XInterrupt, a także określając, czy kanał wymaga utworzenia obiektu urządzenia fizycznego PDO, co w przypadku kanałów VDEV jest zazwyczaj zawsze prawdą. W takim przypadku sterownik VMBus tworzy wystąpienie PDO odzwierciedlającą nowy kanał. Utworzone urządzenie jest chronione przez deskryptor bezpieczeństwa, który sprawia, że jest dostępne tylko z kont systemowych i administracyjnych. Standardowy interfejs urządzenia VMBus, dołączony do nowego PDO, utrzymuje związek pomiędzy nowym kanałem VMBus (poprzez strukturę danych *LOCAL_OFFER*) a obiektem urządzenia. Po utworzeniu PDO menedżer Pnp jest w stanie zidentyfikować i załadować właściwy sterownik VSC poprzez typ VDEV i identyfikatory GUID wystąpienia zawarte w wiadomości *Offer Channel*. Interfejsy te stają się częścią nowego PDO i są widoczne przez menedżera urządzeń (*Device Manager*). Zobacz poniższy eksperyment, aby uzyskać szczegółowe informacje. Kiedy sterownik VSC jest następnie załadowany, zwykle wywołuje API *VmbEnableChannel* (zaprezentowane przez KMCL, jak omówiono wcześniej), aby „otworzyć" kanał i utworzyć ostateczny bufor pierścieniowy.

Eksperyment: wyszczególnienie urządzeń wirtualnych VDEV prezentowanych przez VMBus

Każdy kanał VMBus jest identyfikowany poprzez typ i GUID instancji. Dla kanałów należących do VDEV, typ i GUID instancji identyfikuje również prezentowane urządzenie. Gdy sterownik podrzędny VMBus tworzy instancję PDO, zawiera w niej typ i instancję GUID kanału we właściwościach wielu urządzeń, takich jak ścieżka instancji, ID sprzętu i kompatybilny ID. Ten eksperyment pokazuje, jak wyliczyć wszystkie wirtualne urządzenia VDEV zbudowane na warstwie VMBus.

W celu wykonania tego eksperymentu zbuduj i uruchom maszynę wirtualną Windows 10 poprzez menedżera funkcji Hyper-V (ang. *Hyper-V Manager*). Kiedy maszyna wirtualna jest uruchomiona i działa, otwórz menedżera urządzeń (*Device Manager*), wpisując jego nazwę na przykład w polu wyszukiwania na pasku zadań. W aplecie *Device Manager* kliknij menu *View* (*Widok*) i wybierz opcję *Device by Connection* (*Urządzenia według połączeń*). Sterownik magistrali VMBus jest wyliczany i uruchamiany poprzez moduł wyliczający (ang. *enumerator*) ACPI, dlatego rozwiń węzeł główny *ACPI x64-based PC* (*Komputer oparty na architekturze x64 obsługujący interfejs ACPI*), a następnie *ACPI Module Device* (urządzenie modułu ACPI) znajdujący się w węźle podrzędnym *Microsoft ACPI-Compliant System* (*System zgodny ze standardem Microsoft ACPI*), jak pokazano na poniższym rysunku:

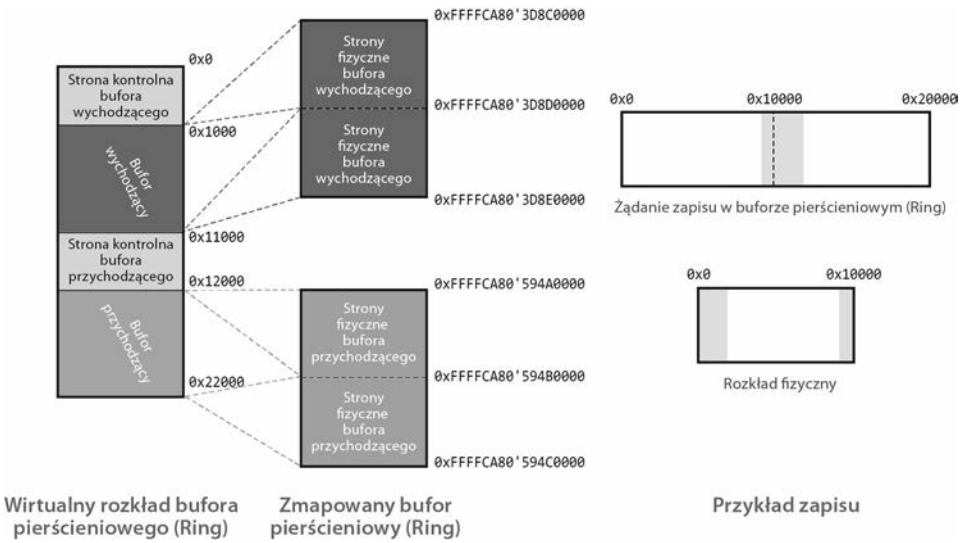


Otwierając *ACPI Module Device*, powinieneś znaleźć kolejny węzeł, nazwany *Microsoft Hyper-V Virtual Machine Bus* (*Dostawca magistrali maszyny wirtualnej funkcji Microsoft Hyper-V*), który odzwierciedla główny obiekt PDO magistrali VMBus. Pod tym węzłem menedżer urządzeń pokazuje wszystkie urządzenia instancji utworzone przez FDO magistrali VMBus po tym jak odpowiednie kanały magistrali VMBus, powiązane z tymi urządzeniami, zostały zaoferowane z poziomu partycji głównej.

Kliknij teraz prawym przyciskiem myszy jedno z urządzeń Hyper-V, takie jak urządzenie *Microsoft Hyper-V Video*, i wybierz opcję *Properties* (*Właściwości*). Aby wyświetlić identyfikatory GUID typu i instancji kanału VMBus wspierającego urządzenie wirtualne, otwórz zakładkę *Details* (*Szczegóły*) okna *Properties* (*Właściwości*). Trzy właściwości urządzenia obejmują typ kanału i identyfikator GUID instancji (wyświetlany w różnych formatach): *Device Instance path* (*Ścieżka wystąpienia urządzenia*), *Hardware Ids* (*Identyfikatory sprzętu*) i *Compatible Ids* (*Zgodne identyfikatory*). Chociaż zgodne identyfikatory zawierają tylko identyfikator GUID typu kanału VMBus (*{da0a7802-e377-4aac-8e77-0558eb1073f8}* na rysunku), identyfikatory sprzętu i ścieżka wystąpienia urządzenia zawierają identyfikatory GUID zarówno typu kanału, jak i jego wystąpienia.

Otwarcie kanału VMBus i utworzenie bufora pierścieniowego

Aby poprawnie uruchomić komunikację między partycjami i utworzyć bufor pierścieniowy, musi zostać otwarty kanał. Zazwyczaj VSC po przydzieleniu kanału po stronie klienta (wciąż poprzez *VmbChannel Allocate*) wywołuje API *VmbChannelEnable* wyeksportowane ze sterownika KMCL. Jak opisano to w poprzednim punkcie, to API w partycjach podrzędnych otwiera kanał VMBus, który został już zaoferowany przez partycję główną (*root*). Sterownik KMCL komunikuje się ze sterownikiem VMBus, uzyskuje parametry kanału (takie jak typ kanału, GUID instancji i używana przestrzeń MMIO) i tworzy element roboczy dla odbieranych pakietów. Następnie przydziela bufor pierścieniowy, który został pokazany na rysunku 9.26. Rozmiar bufora pierścieniowego jest zwykle określany przez VSC poprzez wywołanie eksportowanego przez KMCL interfejsu API *VmbClientChannelInitSetRingBufferPageCount*.



RYSUNEK 9.26. *Przykład 16-stronicowego bufora pierścieniowego przydzielonego w partycji podrzędnej*

Bufor pierścieniowy jest przydzielany z niestronicowanej puli maszyny wirtualnej podrzędnej i mapowany poprzez listę deskryptorów pamięci (MDL — ang. *Memory Descriptor List*) przy użyciu techniki zwanej *podwójnym mapowaniem* (listy MDL zostały opisane w rozdziale 5. części I). W tej technice przydzielony MDL opisuje podwójną liczbę stron fizycznych przychodzącego (lub wychodzącego) bufora. Tablica PFN listy MDL jest wypełniana przez dwukrotne włączenie stron fizycznych bufora: raz w pierwszej połowie tablicy i raz w drugiej. W ten sposób powstaje „bufor pierścieniowy".

Na przykład na rysunku 9.26 bufory przychodzący i wychodzący mają rozmiar 16 stron (0x10). Bufor wychodzący jest zmapowany pod adresem 0xFFCA803D8C0000. Jeśli nadawca zapisze pakiet VMBus o rozmiarze 1 kB w miejscu zbliżonym do końca bufora, powiedzmy na offsecie 0x9FF00, zapis się powiedzie (nie zostanie zgłoszony wyjątek naruszenia dostępu), ale dane zostaną zapisane częściowo na końcu bufora, a częściowo na jego początku. Na rysunku 9.26 tylko 256 (0x100) bajtów jest zapisanych na końcu bufora, natomiast pozostałe 768 (0x300) bajtów na początku.

Zarówno bufory przychodzące, jak i wychodzące otoczone są stroną kontrolną. Strona ta jest współdzielona pomiędzy dwoma punktami końcowymi i składa się na blok kontrolny pierścienia maszyny wirtualnej. Ta struktura danych służy do śledzenia pozycji ostatniego pakietu zapisanego w buforze pierścieniowym. Ponadto zawiera ona kilka bitów kontrolujących, czy wysłać przerwanie, gdy pakiet musi zostać dostarczony.

Po utworzeniu bufora pierścieniowego sterownik KMCL wysyła IOCTL do magistrali VMBus, żądając utworzenia listy deskryptorów GPA (GPADL). GPADL jest strukturą danych bardzo podobną do MDL i służy do opisu kawałka pamięci fizycznej. W odróżnieniu od MDL GPADL zawiera tablicę adresów fizycznych gości (GPA, które zawsze są wyrażone jako liczby 64-bitowe, inaczej niż PFN zawarte w MDL). Sterownik VMBus wysyła różne komunikaty do partycji głównej w celu przeniesienia całego GPADL opisującego zarówno przychodzące, jak i wychodzące bufory pierścieniowe (maksymalny rozmiar syntetycznej wiadomości to 240 bajtów, jak omówiono to wcześniej). Partycja główna rekonstruuje cały GPADL i przechowuje go na wewnętrznej liście. GPADL jest odwzorowywany w partycji głównej (*root*) w momencie, gdy podrzędna maszyna wirtualna wysyła ostateczną wiadomość *Open Channel*. Główny (*root*) sterownik magistrali VMBus parsuje otrzymany GPADL i mapuje go we własnej fizycznej przestrzeni adresowej, korzystając z usług dostarczanych przez sterownik VID (który utrzymuje listę zakresów bloków pamięci składających się na fizyczną przestrzeń adresową maszyny wirtualnej).

Na tym etapie kanał jest gotowy: partycja podrzędna i partycja główna mogą się komunikować, po prostu odczytując lub zapisując dane do bufora pierścieniowego. Gdy nadawca skończy zapisywać swoje dane, wywołuje API *VmbChannelSendSynchronousRequest* prezentowane przez sterownik KMCL. API wywołuje usługi VMBus, aby zasygnalizować zdarzenie na stronie monitora obiektu Xinterrupt związanego z kanałem (stare wersje protokołu VMBus używały strony przerwania, która zawierała bit odpowiadający każdemu kanałowi). Alternatywnie VMBus może zasygnalizować zdarzenie bezpośrednio w porcie zdarzeń kanału, co zależy tylko od wymaganego opóźnienia.

Poza VSC inne komponenty wykorzystują VMBus do implementacji interfejsów wyższego poziomu. Dobrych przykładów dostarczają potoki VMBus, które są zaimplementowane w dwóch bibliotekach trybu jądra (*Vmbuspipe.dll* i *Vmbuspiper.dll*) i polegają na usługach prezentowanych przez sterownik VMBus (poprzez IOCTL). Gniazda platformy Hyper-V (ang. *sockets*; znane również jako HvSockets) umożliwiają szybką komunikację między partycjami przy użyciu standardowych interfejsów sieciowych (gniazd). Klient łączy typ gniazda *AF_HYPERV* z docelową maszyną wirtualną, określając identyfikator GUID docelowej maszyny wirtualnej oraz identyfikator GUID rejestracji usługi gniazda platformy Hyper-V (aby użyć HvSockets, oba punkty końcowe muszą być zarejestrowane w kluczu rejestru *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Virtualization\GuestCommunicationServices*, w którym znajduje się aktualna wirtualizacja) zamiast docelowego adresu IP i portu. Gniazda platformy Hyper-V są zaimplementowane w wielu sterownikach: *HvSocket.sys* jest sterownikiem transportowym, który prezentuje niskopoziomowe usługi używane przez infrastrukturę gniazd; *HvSocketControl.sys* jest sterownikiem kontroli dostawcy używanym do załadowania dostawcy HvSocket, w przypadku gdy interfejs VMBus nie jest obecny w systemie; *HvSocket.dll* jest biblioteką, która prezentuje uzupełniające interfejsy gniazd (związane z gniazdami platformy Hyper-V) wywoływane z aplikacji trybu użytkownika. Opisanie wewnętrznej infrastruktury zarówno gniazd platformy Hyper-V, jak i potoków VMBus wykracza poza zakres tej książki, ale oba te elementy są udokumentowane w Microsoft Docs.

Obsługa sprzętu wirtualnego

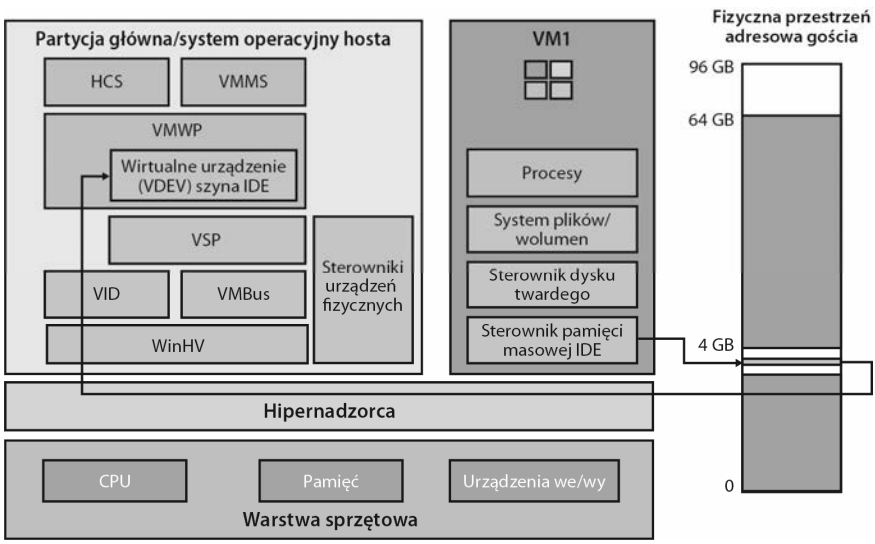
Aby prawidłowo uruchomić maszyny wirtualne, stos wirtualizacji musi obsługiwać urządzenia wirtualne. Platforma Hyper-V obsługuje różne rodzaje urządzeń wirtualnych, które są implementowane w wielu komponentach stosu wirtualizacji. Wejścia i wyjścia urządzeń wirtualnych są obsługiwane przede wszystkim w głównym systemie operacyjnym. Obsługa we/wy obejmuje pamięć masową, sieć, klawiaturę, mysz, porty szeregowe i GPU (procesor graficzny). Stos wirtualizacji udostępnia maszynom wirtualnym gości trzy rodzaje urządzeń:

- urządzenia emulowane, znane również w standardowej formie przemysłowej jako urządzenia w pełni zwirtualizowane;
- urządzenia syntetyczne, znane również jako urządzenia parawirtualizowane;
- urządzenia przyspieszane sprzętowo, znane również jako urządzenia o dostępie bezpośrednim.

W celu wykonania operacji wejścia/wyjścia na urządzeniach fizycznych, procesor zazwyczaj odczytuje i zapisuje dane z portów wejściowych i wyjściowych (portów we/wy), które należą do danego urządzenia. Procesor może uzyskać dostęp do portów we/wy na dwa sposoby:

- Poprzez oddzielną przestrzeń adresową we/wy, która jest odrębna od przestrzeni adresowej pamięci fizycznej i w platformach AMD64 składa się z 64 tysięcy indywidualnie adresowalnych portów we/wy. Ta metoda jest stara i na ogół używana w przypadku urządzeń starszego typu.
- Poprzez we/wy mapowane w pamięci. Urządzenia, które reagują jak elementy pamięci, mogą być dostępne poprzez przestrzeń adresową pamięci fizycznej procesora. Oznacza to, że procesor uzyskuje dostęp do pamięci za pomocą standardowych instrukcji: bazowa pamięć fizyczna jest mapowana do tego urządzenia.

Rysunek 9.27 przedstawia przykład emulowanego urządzenia (wirtualny kontroler IDE używany w maszynach wirtualnych pierwszej generacji), które wykorzystuje mapowane w pamięci operacje we/wy do przesyłania danych do i z wirtualnego procesora.



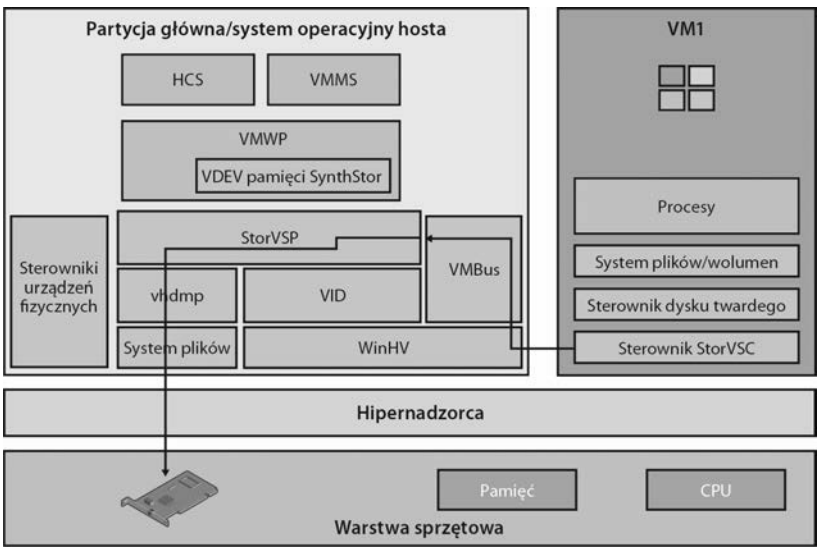
RYSUNEK 9.27. *Wirtualny kontroler IDE, który wykorzystuje emulowane wejścia/wyjścia do wykonywania transferu danych*

W tym modelu za każdym razem, gdy wirtualny procesor odczytuje lub zapisuje do przestrzeni MMIO urządzenia lub emituje instrukcje dostępu do portów we/wy, wywołuje VMEXIT do hipernadzorcy. Hipernadzorca wywołuje odpowiedni program przechwytujący, który jest wysyłany do sterownika VID. Sterownik VID buduje wiadomość VID i umieszcza ją w wewnętrznej kolejce. Kolejka ta jest opróżniana przez wewnętrzny wątek VMWP, który oczekuje i wysyła komunikaty VP otrzymane od sterownika VID; wątek ten jest nazywany wątkiem *pompy komunikatów* i należy do wewnętrznej puli wątków zainicjalizowanej w czasie tworzenia VMWP. Proces roboczy maszyny wirtualnej identyfikuje adres fizyczny powodujący VMEXIT, który jest związany z właściwym urządzeniem wirtualnym (VDEV), i wywołuje jedno z wywołań zwrotnych urządzenia wirtualnego VDEV (zwykle wywołanie zwrotne „odczyt" lub „zapis"). Kod VDEV korzysta z usług dostarczanych przez emulator instrukcji, aby wykonać instrukcję powodującą błąd i prawidłowo emulować urządzenie wirtualne (na przykładzie kontrolera IDE).

Uwaga. Emulator pełnych instrukcji znajdujący się w procesie roboczym maszyny wirtualnej jest również wykorzystywany do innych różnych celów, takich jak przyspieszenie przypadków przechwytywania intensywnego kodu w partycji podrzędnej. Emulator w tym przypadku pozwala na pozostanie kontekstu wykonawczego w procesie roboczym maszyny wirtualnej pomiędzy przechwyceniami, ponieważ instrukcje VMEXIT mają poważny narzut wydajności. Starsze wersje rozszerzeń wirtualizacji sprzętu zabraniają wykonywania kodu w trybie rzeczywistym w maszynie wirtualnej; w tych przypadkach stos wirtualizacji używał emulatora do wykonywania kodu w trybie rzeczywistym w maszynie wirtualnej.

Urządzenia parawirtualizowane

Podczas gdy urządzenia emulowane zawsze powodują VMEXIT i są dość powolne, rysunek 9.28 pokazuje przykład urządzenia syntetycznego lub parawirtualizowanego: syntetyczny adapter pamięci masowej. Urządzenia syntetyczne wiedzą, że działają w środowisku zwirtualizowanym; zmniejsza to złożoność urządzenia wirtualnego i pozwala mu osiągnąć wyższą wydajność. Niektóre syntetyczne urządzenia wirtualne istnieją tylko w postaci wirtualnej i nie emulują żadnego rzeczywistego sprzętu fizycznego (przykładem jest syntetyczny protokół RDP).



RYSUNEK 9.28. *Urządzenie parawirtualizowane z kontrolerem pamięci masowej*

Urządzenia parawirtualizowane wymagają na ogół trzech głównych komponentów, takich jak:

- Sterownik dostawcy usług wirtualizacji VSP, który działa w partycji głównej i prezentuje gościowi interfejsy specyficzne dla wirtualizacji dzięki usługom dostarczanym przez VMBus (szczegóły na temat VMBus znajdują się w poprzednim rozdziale).
- Syntetyczny VDEV, który jest mapowany w procesie roboczym maszyny wirtualnej i zazwyczaj współpracuje tylko przy uruchamianiu, usuwaniu, zapisywaniu i przywracaniu urządzenia wirtualnego. Sterownik ten na ogół nie jest wykorzystywany podczas regularnej pracy urządzenia. Syntetyczne urządzenie VDEV inicjalizuje i przydziela zasoby specyficzne dla urządzenia (w tym przykładzie SynthStor VDEV inicjalizuje adapter wirtualnej pamięci masowej), ale przede wszystkim pozwala dostawcy usług wirtualizacji (VSP) zaoferować kanał komunikacyjny VMBus dla klienta usługi wirtualizacji (VSC) gościa. Kanał ten będzie wykorzystywany do komunikacji z partycją główną (*root*) oraz do sygnalizacji powiadomień, specyficznych dla urządzenia, przez hipernadzorcę.
- Sterownik konsumenta usługi wirtualizacji (VSC), który działa w partycji podrzędnej, rozumie specyficzne dla wirtualizacji interfejsy prezentowane przez VSP i odczytuje/zapisuje wiadomości i powiadomienia z pamięci współdzielonej prezentowanej za pośrednictwem VMBus przez VSP. Dzięki temu urządzenie wirtualne może działać w maszynie wirtualnej podrzędnej szybciej niż urządzenie emulowane.

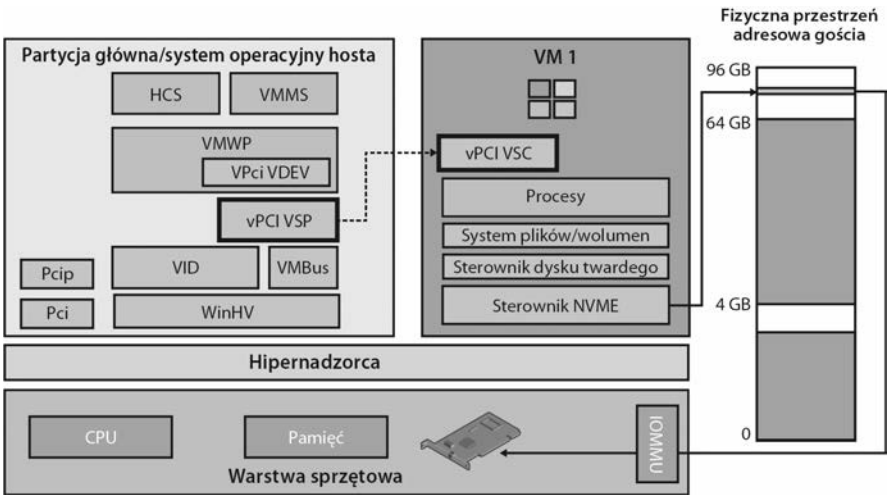
Urządzenia przyspieszane sprzętowo

W serwerowych jednostkach SKU urządzenia *przyspieszane sprzętowo* (znane również jako urządzenia o dostępie bezpośrednim) pozwalają na ponowne mapowanie urządzeń fizycznych w partycji gościa dzięki usługom prezentowanym przez infrastrukturę VPCI. Gdy urządzenie fizyczne obsługuje technologie takie jak wirtualizacja wejścia/wyjścia z jednym rootem (SR-IOV — ang. *single-root input/output virtualization*) lub przypisywanie urządzeń dyskretnych (DDA — ang. *discrete device assignment*), można je mapować do partycji gościa. Partycja gościa ma bezpośredni dostęp do przestrzeni MMIO związanej z urządzeniem i może wykonywać DMA do i z pamięci gościa bezpośrednio, bez przechwytywania przez hipernadzorcę. IOMMU zapewnia potrzebne bezpieczeństwo i gwarantuje, że urządzenie może inicjalizować transfery DMA tylko w pamięci fizycznej należącej do maszyny wirtualnej.

Rysunek 9.29 przedstawia komponenty odpowiedzialne za zarządzanie urządzeniami przyspieszanymi sprzętowo:

- VPci VDEV (*Vpcievdev.dll*) działa w procesie roboczym maszyny wirtualnej. Jego zasadą jest wyodrębnienie listy urządzeń przyspieszanych sprzętowo z pliku konfiguracyjnego maszyny wirtualnej, skonfigurowanie wirtualnej magistrali VPCI i przypisanie urządzenia do VSP.
- Sterownik PCI Proxy (*Pcip.sys*) odpowiada za odmontowanie i zamontowanie urządzenia fizycznego zgodnego z przypisaniem urządzenia dyskretnego (DDA) z poziomu partycji głównej. Ponadto pełni on kluczową rolę w uzyskiwaniu listy zasobów wykorzystywanych przez urządzenie (poprzez wirtualizację SR-IOV), takich jak przestrzeń MMIO i przerwania. Sterownik proxy zapewnia dostęp do fizycznej przestrzeni konfiguracyjnej urządzenia i sprawia, że „niezamontowane” urządzenie jest niedostępne dla systemu operacyjnego hosta.

- Dostawca usługi wirtualnej VPCI (*Vpcivsp.sys*) tworzy i utrzymuje obiekt magistrali wirtualnej, powiązany z jednym lub większą liczbą urządzeń przyspieszanych sprzętowo (które po stronie dostawcy usługi wirtualnej VPCI są nazywane *urządzeniami wirtualnymi*). Urządzenia wirtualne są prezentowane maszynie wirtualnej gościa poprzez kanał VMBus utworzony przez dostawcę usługi wirtualnej VSP i oferowany klientowi usługi wirtualnej VSC w partycji gościa.
- Klient usługi wirtualnej VPCI (Vpci.sys) jest sterownikiem magistrali WDF, który działa w maszynie wirtualnej gościa. Klient ten łączy się z kanałem VMBus prezentowanym przez VSP, otrzymuje listę urządzeń dostępu bezpośredniego zaprezentowanych maszynie wirtualnej oraz ich zasoby, a następnie tworzy obiekt fizycznego urządzenia (PDO — ang. *physical device object*) dla każdego z nich. Sterownik urządzeń może następnie dołączyć do utworzonych PDO w taki sam sposób, jak w środowiskach niewirtualizowanych.



RYSUNEK 9.29. *Urządzenia przyspieszane sprzętowo*

Kiedy użytkownik chce zmapować urządzenie przyspieszane sprzętowo do maszyny wirtualnej, używa pewnych poleceń powłoki PowerShell (więcej szczegółów w poniższym eksperymencie), które rozpoczynają się od „odmontowania" urządzenia z partycji głównej (*root*). To działanie zmusza usługę VMMS do komunikacji ze standardowym sterownikiem PCI (poprzez jego zaprezentowane urządzenie o nazwie *PciControl*). Usługa VMMS wysyła IOCTL *PCIDRIVE_ADD_VMPROXYPATH* do sterownika PCI, przekazując deskryptor urządzenia (w postaci identyfikatora magistrali, urządzenia i funkcji). Sterownik PCI sprawdza deskryptor i jeśli weryfikacja przebiegła pomyślnie, dodaje go do wartości rejestru *HKLM\System\CurrentControlSet\Control\PnP\Pci\VmProxy*. Następnie VMMS rozpoczyna (ponowne) wyliczanie urządzenia PNP za pomocą usług udostępnianych przez menedżera PNP. W fazie wyliczania sterownik PCI znajduje nowe urządzenie proxy i ładuje sterownik PCI proxy (*Pcip.sys*), który oznacza urządzenie jako zarezerwowane dla stosu wirtualizacji i czyni je niewidocznym dla systemu operacyjnego hosta.

Drugi krok wymaga przypisania urządzenia do maszyny wirtualnej. W tym przypadku VMMS zapisuje deskryptor urządzenia w pliku konfiguracyjnym VM. Po uruchomieniu maszyny wirtualnej, urządzenie wirtualne VPCI (VPCI VDEV) (*vpcievdev.dll*) odczytuje deskryptor

urządzenia o dostępie bezpośrednim z konfiguracji maszyny wirtualnej i rozpoczyna złożoną fazę konfiguracji, która jest sterowana głównie przez Dostawcę usługi wirtualnej VPCI (VPCI VSP) (*Vpcivsp.sys*). W rzeczywistości w swoim wywołaniu zwrotnym „zasilanie włączone" VPCI VDEV wysyła różne IOCTL do Dostawcy usługi wirtualnej VPCI (który działa w partycji głównej) w celu wykonania utworzenia wirtualnej magistrali i przypisania urządzeń przyspieszanych sprzętowo do maszyny wirtualnej gościa.

„Wirtualna szyna" jest strukturą danych używaną przez infrastrukturę VPCI jako „klej" do utrzymania połączenia między partycją główną, maszyną wirtualną gościa i przypisanymi do niej urządzeniami dostępu bezpośredniego. Dostawca usługi wirtualnej VPCI (VPCI VSP) przydziela i uruchamia kanał VMBus oferowany maszynie wirtualnej gościa i zamyka go w wirtualnej szynie. Ponadto magistrala wirtualna zawiera pewne wskaźniki do ważnych struktur danych, takich jak niektóre przydzielone pakiety VMBus używane do komunikacji dwukierunkowej, stan zasilania gościa itp. Po utworzeniu wirtualnej magistrali VPCI VSP wykonuje przypisanie urządzenia.

Urządzenie przyspieszane sprzętowo jest wewnętrznie identyfikowane przez lokalne unikalne ID (LUID — ang. *local unique ID*) i jest odzwierciedlane przez obiekt urządzenia wirtualnego, który jest przydzielany przez VPCI VSP. Na podstawie LUID urządzenia VPCI VSP lokalizuje właściwy sterownik proxy, który jest również znany jako sterownik Mux — zwykle jest to *Pcip.sys*. VPCI VSP odpytuje interfejs wirtualizacji SR-IOV lub interfejs przypisania urządzenia dyskretnego (DDA) ze sterownika proxy i używa ich do uzyskania informacji *Plug and Play* (deskryptora sprzętowego) urządzenia o bezpośrednim dostępie oraz do zebrania wymagań dotyczących zasobów (przestrzeń MMIO, rejestry BAR i kanały DMA). W tym momencie urządzenie jest gotowe do podłączenia do maszyny wirtualnej gościa: VPCI VSP wykorzystuje usługi prezentowane przez sterownik WinHvr do wysłania hiperwywołania *HvAttachDevice* do hipernadzorcy, który ponownie konfiguruje systemową jednostkę IOMMU do mapowania przestrzeni adresowej urządzenia w partycji gościa.

Maszyna wirtualna gościa jest świadoma obecności zmapowanego urządzenia dzięki VPCI VSC (*Vpci.sys*). VPCI VSC jest sterownikiem magistrali WDF wyliczanym i uruchamianym przez sterownik magistrali VMBus znajdujący się w maszynie wirtualnej gościa. Składa się on z dwóch głównych komponentów: obiektu jako urządzenia funkcjonalnego (FDO — ang. *Functional Device Object*), tworzonego podczas startu maszyny wirtualnej, oraz jednego lub kilku obiektów jako urządzeń fizycznych (PDO — ang. *Physical Device Objects*), odzwierciedlających fizyczne urządzenia dostępu bezpośredniego ponownie mapowane w maszynie wirtualnej gościa. Kiedy sterownik magistrali VPCI VSC jest wykonywany w maszynie wirtualnej gościa, tworzy i uruchamia część kliencką kanału VMBus używaną do wymiany wiadomości z VSP-em. *Send bus relations* to pierwszy komunikat wysyłany przez VPCI VSC przez kanał VMBus. Dostawca usług wirtualizacji VSP w partycji głównej odpowiada, wysyłając listę identyfikatorów sprzętowych opisujących urządzenia przyspieszane sprzętowo, które są aktualnie dołączone do maszyny wirtualnej. Gdy menedżer PNP zażąda nowych relacji urządzeń do klienta usługi wirtualizacji magistrali VPCI (VPCI VSC), ten ostatni tworzy nowy PDO dla każdego odkrytego urządzenia z dostępem bezpośrednim. Sterownik klienta VSC wysyła kolejną wiadomość do dostawcy VSP w celu zażądania zasobów używanych przez PDO.

Po przeprowadzeniu wstępnej konfiguracji VSC i VSP rzadko angażują się w zarządzanie urządzeniami. Sterownik konkretnego urządzenia przyspieszanego sprzętowo w maszynie wirtualnej gościa dołącza do odpowiedniego PDO i zarządza urządzeniem peryferyjnym tak, jakby było ono zainstalowane na maszynie fizycznej.

Eksperyment: mapowanie dysku NVMe przyspieszanego sprzętowo do maszyny wirtualnej

Jak wyjaśniono w poprzednim punkcie, urządzenia fizyczne obsługujące technologie SR-IOV i DDE mogą być bezpośrednio mapowane w maszynie wirtualnej gościa uruchomionej w hoście Windows Server 2019. W tym eksperymencie mapujemy dysk NVMe, który jest podłączony do systemu przez magistralę PCI-Ex i obsługuje DDE, do maszyny wirtualnej Windows 10. (Windows Server 2019 obsługuje również bezpośrednie przypisanie karty graficznej, ale jest to poza zakresem tego eksperymentu).

Jak wyjaśniono na stronie *https://docs.microsoft.com/en-us/virtualization/community/team-blog/2015/20151120-discrete-device-assignment-machines-and-devices*, aby urządzenie mogło być ponownie przypisane, powinno mieć pewne właściwości, takie jak obsługa przerwań sygnalizowanych komunikatami i we/wy mapowanych w pamięci. Ponadto maszyna, na której działa hipernadzorca, powinna obsługiwać wirtualizację SR-IOV i posiadać odpowiednią jednostkę zarządzania pamięcią we/wy (I/O MMU). W przypadku tego eksperymentu należy zacząć od sprawdzenia, czy standard SR-IOV jest włączony w systemowym BIOS-ie (nie wyjaśniono tego tutaj, procedura różni się w zależności od producenta maszyny).

Kolejnym krokiem jest pobranie skryptu PowerShell, który weryfikuje, czy kontroler NVMe jest zgodny z przypisaniem urządzenia dyskretnego (DDA). Pobierz skrypt PowerShell *survey-dda.ps1* ze strony *https://github.com/MicrosoftDocs/Virtualization-Documentation/tree/master/hyperv-samples/benarm-powershell/DDA*. Otwórz administracyjne okno PowerShell (wpisz **`PowerShell`** w polu wyszukiwania na pasku zadań, a następnie kliknij prawym przyciskiem myszy ikonę *PowerShell* i wybierz opcję *Run as administrator* (*Uruchom jako administrator*)) i uruchamiając polecenie `Get-ExecutionPolicy`, sprawdź, czy zasada wykonywania skryptów PowerShella jest ustawiona na nieograniczoną. Jeśli polecenie da jakieś dane wyjściowe inne niż `Unrestricted`, wpisz następujące polecenie: **`Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy Unrestricted`**, naciśnij *Enter* i potwierdź klawiszem *Y*.

Jeśli wykonasz pobrany skrypt *survey-dda.ps1*, wyświetlone przez niego dane wyjściowe powinny pokazać, czy urządzenie NVMe może zostać ponownie przypisane do maszyny wirtualnej gościa. Oto poprawny przykład danych wyjściowych:

```
Standard NVM Express Controller
Express Endpoint -- more secure.
    And its interrupts are message-based, assignment can work.
PCIROOT(0)#PCI(0302)#PCI(0000)
```

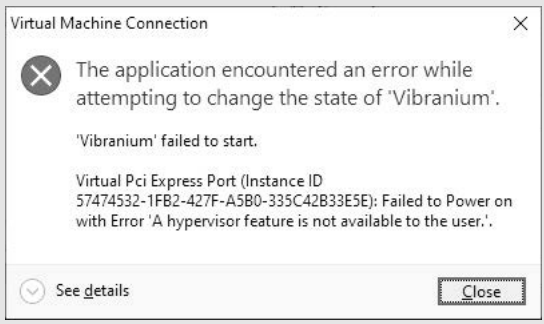
Zwróć uwagę na ścieżkę lokalizacji (w tym przykładzie ciąg *PCIROOT(0)#PCI(0302)#PCI(0000)*). Teraz ustawimy akcję automatycznego zatrzymania dla docelowej maszyny wirtualnej jako wyłączoną (krok wymagany dla DDA) i odmontujemy urządzenie. W naszym przykładzie maszyna wirtualna nosi nazwę *Vibranium*. Napisz następujące polecenia w oknie PowerShella (zastępując przykładową nazwę VM i lokalizację urządzenia własnymi):

```
Set-VM -Name "Vibranium" -AutomaticStopAction TurnOff
Dismount-VMHostAssignableDevice -LocationPath "PCIROOT(0)#PCI(0302)#PCI(0000)"
```

W przypadku gdy ostatnie polecenie przyniesie błąd nieudanej operacji, to prawdopodobnie nie wyłączyłeś urządzenia. Otwórz menedżera urządzeń, zlokalizuj swój kontroler NVMe (w tym przykładzie jest to Standard NVMe Express Controller), kliknij go prawym przyciskiem myszy i wybierz *Disable Device*. Następnie możesz ponownie wpisać ostatnie polecenie. Tym razem powinno się to udać. Następnie przypisz urządzenie do swojej maszyny wirtualnej, wpisując następujące polecenia:

```
Add-VMAssignableDevice -LocationPath "PCIROOT(0)#PCI(0302)#PCI(0000)" -VMName "Vibranium"
```

Ostatnie polecenie powinno całkowicie usunąć kontroler NVMe z hosta. Należy to sprawdzić w menedżerze urządzeń w systemie hosta. Teraz nadszedł czas, aby włączyć zasilanie maszyny wirtualnej. Możesz użyć menedżera funkcji Hyper-V (ang. *Hyper-V Manager*) lub PowerShell. Jeśli po uruchomieniu maszyny wirtualnej pojawi się błąd taki jak poniżej, oznacza to, że BIOS nie jest odpowiednio skonfigurowany do prezentowania SR-IOV lub że jednostka zarządzania pamięcią we/wy (I/O MMU) nie posiada wymaganych właściwości (najprawdopodobniej nie obsługuje ponownego mapowania we/wy).



W przeciwnym razie maszyna wirtualna powinna wykonać rozruch (*boot*) zgodnie z oczekiwaniami. W tym przypadku kontroler NVMe i dysk NVMe powinny być widoczne w aplecie Device Manager maszyny wirtualnej podrzędnej. Za pomocą narzędzia do zarządzania dyskami można utworzyć partycje w maszynie wirtualnej podrzędnej w taki sam sposób jak w systemie operacyjnym hosta. Dysk NVMe będzie pracował z pełną prędkością i nie będzie miał problemów z wydajnością (można to potwierdzić za pomocą dowolnego narzędzia do sprawdzania wydajności dysków).

Aby prawidłowo usunąć urządzenie z maszyny wirtualnej i ponownie zamontować je w systemie operacyjnym hosta, najpierw zamknij maszynę wirtualną, a następnie użyj następujących poleceń (pamiętaj, aby zawsze zmieniać nazwę maszyny wirtualnej i lokalizację kontrolera NVMe):

```
Remove-VMAssignableDevice -LocationPath "PCIROOT(0)#PCI(0302)#PCI(0000)" -VMName "Vibranium"
Mount-VMHostAssignableDevice -LocationPath "PCIROOT(0)#PCI(0302)#PCI(0000)"
```

Po wykonaniu ostatniego polecenia kontroler NVMe powinien ponownie pojawić się na liście w Menedżerze urządzeń systemu operacyjnego hosta. Wystarczy go ponownie włączyć przy ponownym uruchomieniu, aby móc korzystać z dysku NVMe w hoście.

Maszyny wirtualne wspierane przez VA

Maszyny wirtualne są wykorzystywane do wielu celów. Jednym z nich jest właściwe uruchamianie tradycyjnego oprogramowania w odizolowanych środowiskach, zwanych *kontenerami*. (Silosy serwerów i aplikacji, które są dwoma rodzajami kontenerów, zostały wprowadzone w części I, w rozdziale 3., „Procesy i zadania"). W pełni izolowane kontenery (wewnętrznie nazwane Xenon i Krypton) wymagają typu szybkiego startu, niskiego narzutu oraz możliwości uzyskania jak najmniejszego śladu pamięciowego. Pamięć fizyczna gościa w tego typu maszynie wirtualnej jest zazwyczaj współdzielona pomiędzy wieloma kontenerami. Dobrych przykładów kontenerów dostarcza mechanizm Windows Defender Application Guard, który wykorzystuje kontener do zapewnienia pełnej izolacji przeglądarki, lub Windows Sandbox, który wykorzystuje kontenery do zapewnienia w pełni izolowanego środowiska wirtualnego. Zazwyczaj kontener współdzieli to samo oprogramowanie układowe firmware maszyny wirtualnej, system operacyjny, a często także niektóre aplikacje w niej uruchomione (współdzielone komponenty składają się na warstwę bazową kontenera). Uruchomienie każdego kontenera w jego prywatnej przestrzeni pamięci fizycznej gościa byłoby niewykonalne i skutkowałoby dużym marnotrawstwem pamięci fizycznej.

Aby rozwiązać ten problem, stos wirtualizacji zapewnia obsługę maszyn wirtualnych wspieranych przez VA. Maszyny wirtualne ze wsparciem adresów wirtualnych (VA) wykorzystują menedżera pamięci systemu operacyjnego hosta do udostępnienia na rzecz pamięci fizycznej partycji gościa zaawansowanych funkcji, takich jak usuwanie duplikacji pamięci, przycinanie pamięci, mapy bezpośrednie, klonowanie pamięci i co najważniejsze, stronicowanie (wszystkie te koncepcje zostały szeroko omówione w rozdziale 5. części I). W przypadku tradycyjnych maszyn wirtualnych pamięć gościa jest przydzielana przez sterownik VID poprzez statyczne przydzielanie systemowych stron fizycznych z hosta i mapowanie ich w przestrzeni GPA maszyny wirtualnej, zanim jakikolwiek wirtualny procesor będzie miał szansę na wykonanie operacji. Zamiast mapować strony SPA bezpośrednio do przestrzeni GPA, VID tworzy przestrzeń GPA, która jest początkowo pusta, tworzy minimalny proces trybu użytkownika (zwany VMMEM) dla hostingu przestrzeni adresów wirtualnych i ustawia mapowania GPA do VA za pomocą wirtualnej mikromaszyny (MicroVM). MicroVM jest nowym składnikiem jądra NT ściśle zintegrowanym z menedżerem pamięci NT, który jest ostatecznie odpowiedzialny za zarządzanie mapowaniem GPA do SPA poprzez łączenie mapowania GPA do VA (utrzymywanego przez VID) z mapowaniem VA do SPA (utrzymywanym przez menedżera pamięci NT).

Nowa warstwa pośredniczenia pozwala maszynom wirtualnym obsługiwanym przez VA korzystać z większości funkcji zarządzania pamięcią, które są prezentowane procesom Windows. Jak omówiono to w poprzednim punkcie, proces roboczy maszyny wirtualnej, uruchamiając maszynę wirtualną, prosi sterownik VID o utworzenie bloku pamięci partycji. W przypadku gdy maszyna wirtualna obsługuje adresy VA, tworzy ona bitmapę mapowania zakresu bloku pamięci GPA, która służy do śledzenia przydzielonych stron wirtualnych wspierających pamięć RAM nowej maszyny wirtualnej. Następnie tworzy pamięć RAM partycji, wspieraną przez duży zakres przestrzeni VA. Przestrzeń VA jest zwykle tak duża jak przydzielona ilość pamięci RAM maszyny wirtualnej (zauważ, że nie jest to warunek konieczny: różne zakresy VA mogą być mapowane jako różne zakresy GPA) i jest zarezerwowana w kontekście procesu VMMEM za pomocą natywnego API *NtAllocateVirtualMemory*.

Jeśli optymalizacja „odroczone zadeklarowanie" (ang. *deferred commit*) nie jest włączona (więcej szczegółów w następnej sekcji), sterownik VID wykonuje kolejne wywołanie API *NtAllocateVirtualMemory* z zamiarem zadeklarowania całego zakresu VA. Jak omówiono w rozdziale 5. części I, zadeklarowanie (ang. *commit*) pamięci obciąża systemowy limit przydzieleń, ale nadal nie przydziela żadnej fizycznej strony (wszystkie wpisy PTE opisujące cały zakres

są nieważnymi PTE-ami typu zerowe żądanie (ang. *demand-zero*)). Sterownik VID na tym etapie używa Winhvr, aby poprosić hipernadzorcę o zmapowanie całej przestrzeni GPA partycji do specjalnego, nieważnego SPA (używając tego samego hiperwywołania *HvMapGpaPages* używanego w przypadku standardowych partycji). Kiedy partycja gościa uzyskuje dostęp do pamięci fizycznej gościa, która jest zmapowana w tablicy SLAT przez specjalne, nieprawidłowe SPA, powoduje VMEXIT do hipernadzorcy, który rozpoznaje specjalną wartość i wstrzykuje przechwyt pamięci do partycji głównej (*root*).

Sterownik VID ostatecznie powiadamia MicroVM o nowym zakresie GPA obsługiwanym przez VA, wywołując program *VmCreateMemoryRange* (usługi MicroVM są prezentowane przez jądro NT sterownikowi VID poprzez Rozszerzenie jądra (ang. *Kernel Extension*)). MicroVM przydziela i inicjalizuje strukturę danych *VM_PROCESS_CONTEXT*, która zawiera dwa ważne drzewa RB: jedno opisujące przydzielone zakresy GPA w maszynie wirtualnej i jedno opisujące odpowiadające im zakresy systemowych adresów wirtualnych SVA (ang. *system virtual address*) w partycji głównej. Wskaźnik do przydzielonej struktury danych jest następnie przechowywany w strukturze danych EPROCESS wystąpienia VMMEM.

Gdy proces roboczy maszyny wirtualnej chce dokonać zapisu do pamięci maszyny wirtualnej wspieranej przez VA lub gdy zostanie wygenerowane przechwycenie pamięci z powodu nieprawidłowej translacji GPA do SPA, sterownik VID wywołuje obsługę błędu strony wirtualnej mikromaszyny MicroVM (*VmAccessFault*). Handler wykonuje dwie ważne operacje: najpierw rozwiązuje błąd poprzez wstawienie poprawnego PTE do tablicy stron opisującej błędną stronę wirtualną (więcej szczegółów w rozdziale 5. części I), a następnie aktualizuje tablicę SLAT pochodnej maszyny wirtualnej (poprzez wywołanie sterownika WinHvr, który emituje kolejne hiperwywołanie *HvMapGpaPages*). Następnie strony fizyczne maszyny wirtualnej gościa mogą być stronicowane, ponieważ prywatna pamięć procesu jest normalnie stronicowalna. Ma to te ważne następstwa, że wymaga, aby większość funkcji MicroVM działała w pasywnym IRQL.

Wiele usług menedżera pamięci NT może być wykorzystywanych w maszynach wirtualnych wspieranych przez VA. W szczególności *szablony klonów* pozwalają na szybkie sklonowanie pamięci dwóch różnych maszyn wirtualnych ze wsparciem VA; *bezpośrednie mapowanie* umożliwia współdzielonym obrazom wykonywalnym lub plikom danych mapowanie ich obiektów należących do sekcji do procesu VMMEM i do zakresu GPA wskazującego na dany region VA. Bazowe strony fizyczne mogą być współdzielone pomiędzy różnymi maszynami wirtualnymi i procesami hosta, co prowadzi do poprawy gęstości pamięci.

Optymalizacje maszyn wirtualnych wspieranych przez VA

Jak opisano to w poprzedniej sekcji, koszt dostępu gościa do dynamicznie wspieranej pamięci, która nie jest aktualnie wspierana lub nie przyznaje wymaganych uprawnień, może być dość pokaźny: w przypadku próby dostępu gościa do niedostępnej pamięci następuje VMEXIT, co wymaga od hipernadzorcy zawieszenia wirtualnego procesora gościa, zaplanowania wirtualnego procesora partycji głównej i wstrzyknięcia do niego (procesora) komunikatu o przechwyceniu pamięci. Handler wywołania zwrotnego przechwytu sterownika VID jest wywoływany na wysokim IRQL, ale przetwarzanie żądania i wywołanie w MicroVM wymaga działania na poziomie *PASSIVE_LEVEL*. W związku z tym w kolejce ustawiane jest wywołanie odroczonej procedury (DPC — ang. *Deferred Procedure Call*). Program DPC ustawia zdarzenie, które budzi odpowiedni wątek odpowiedzialny za przetwarzanie przechwycenia. Po tym jak mechanizm obsługi błędów stron MicroVM usunie błąd i wywoła hipernadzorcę, aby zaktualizować wpis SLAT (poprzez kolejne hiperwywołanie, które powoduje kolejny VMEXIT), mechanizm ten wznawia działanie wirtualnego procesora gościa.

Duża liczba przechwyceń pamięci generowanych w czasie rzeczywistym powoduje duże kary za wydajność. Aby tego uniknąć, zaimplementowano wiele optymalizacji w postaci rozszerzeń „oświecających" maszynę wirtualną gościa (lub prostych konfiguracji):

- „oświecenia" związane z zerowaniem pamięci,
- podpowiedzi dotyczące dostępu do pamięci,
- „oświecone" błędy stron,
- odroczone zadeklarowanie (ang. *deferred commit*) i inne optymalizacje.

„Oświecenia" związane z zerowaniem pamięci

Aby uniknąć ujawnienia maszynie wirtualnej informacji o artefaktach pamięci używanych wcześniej przez partycję główną lub inną maszynę wirtualną, pamięć RAM gościa jest zerowana przed mapowaniem jej w celu uzyskania dostępu przez gościa. Zazwyczaj system operacyjny zeruje całą pamięć fizyczną podczas uruchamiania, ponieważ w systemie fizycznym jej zawartość jest niedeterministyczna. W przypadku maszyny wirtualnej oznacza to, że pamięć może zostać wyzerowana dwukrotnie: raz przez hosta wirtualizacji i ponownie przez system operacyjny gościa. W przypadku maszyn wirtualnych ze wsparciem *fizycznym* jest to w najlepszym wypadku marnowanie cykli procesora. W przypadku maszyn wirtualnych ze wsparciem VA zerowanie przez system operacyjny gościa generuje kosztowne przechwytywanie pamięci. Aby uniknąć marnowania przechwytów, hipernadzorca prezentuje rozszerzenia „oświecające" zerowanie pamięci.

Kiedy Windows Loader ładuje główny system operacyjny, używa usług dostarczanych przez oprogramowanie układowe firmware UEFI, aby uzyskać mapę pamięci fizycznej maszyny. Kiedy hipernadzorca uruchamia maszynę wirtualną ze wsparciem VA, prezentuje hiperwywołanie *HvGetBootZeroedMemory*, którego Windows Loader może użyć do odpytywania listy zakresów pamięci fizycznej, które faktycznie są już wyzerowane. Przed przekazaniem wykonania do jądra NT, Windows Loader łączy uzyskane wyzerowane zakresy z listą deskryptorów pamięci fizycznej uzyskanych za pośrednictwem usług EFI i przechowywanych w bloku Loader (dalsze szczegóły dotyczące mechanizmów uruchamiania dostępne są w rozdziale 12.). Jądro NT wstawia scalony deskryptor bezpośrednio na listę wyzerowanych stron, pomijając początkowe zerowanie pamięci.

W podobny sposób hipernadzorca obsługuje rozszerzenie „oświecające" pozwalające na zerowanie pamięci „na gorąco" za pomocą prostej implementacji: Kiedy sterownik VSC pamięci dynamicznej (*dmvsc.sys*) inicjalizuje żądanie dodania pamięci fizycznej do jądra NT, określa flagę `MM_ADD_PHYSICAL_MEMORY_ALREADY_ZEROED`, która podpowiada menedżerowi pamięci (MM), aby dodał nowe strony bezpośrednio do listy stron wyzerowanych.

Podpowiedzi dotyczące dostępu do pamięci

W przypadku maszyn wirtualnych z *fizycznym* wsparciem partycja główna posiada bardzo ograniczone informacje o tym, jak menedżer pamięci gościa zamierza wykorzystać swoje strony fizyczne. W przypadku tych maszyn wirtualnych informacje te są w większości przypadków nieistotne, ponieważ prawie wszystkie mapowania pamięci i GPA są tworzone podczas uruchamiania maszyny wirtualnej i pozostają statycznie zmapowane. W przypadku maszyn wirtualnych ze wsparciem VA informacje te mogą być bardzo przydatne, ponieważ menedżer pamięci hosta zarządza zbiorem roboczym minimalnego procesu zawierającego pamięć maszyny wirtualnej (VMMEM).

Podpowiedź „na gorąco” pozwala gościowi wskazać, że zestaw stron fizycznych powinien zostać zmapowany do gościa, ponieważ będą one dostępne wkrótce lub często. Oznacza to, że strony są dodawane do zestawu roboczego procesu minimalnego. Sterownik VID obsługuje podpowiedź, mówiąc wirtualnej mikromaszynie MicroVM, aby natychmiast oznaczyła strony fizyczne jako błędne i nie usuwała ich ze zbioru roboczego procesu VMMEM.

W podobny sposób podpowiedź „na zimno” pozwala gościowi wskazać, że z zestawu stron fizycznych powinno zostać usunięte mapowanie do gościa, ponieważ zestaw ten nie będzie używany w najbliższym czasie. Sterownik VID obsługuje podpowiedź, przekazując ją do MicroVM, która natychmiast usuwa strony ze zbioru roboczego. Zazwyczaj gość używa zimnej podpowiedzi dla stron, które zostały wyzerowane przez wątek zerowania stron w tle (więcej szczegółów w rozdziale 5. części I).

Partycja gościa ze wsparciem VA określa podpowiedź pamięciową dla strony za pomocą hiperwywołania *HvMemoryHeatHint*.

Błąd strony „oświeconej” (EPF)

Błąd strony „oświeconej” (EPF) to funkcja, która pozwala partycji gościa ze wsparciem VA na zmianę harmonogramu wątków na procesorze wirtualnym, który spowodował przechwycenie pamięci dla strony GPA wspieranej przez VA. Normalnie przechwycenie pamięci dla takiej strony jest obsługiwane przez synchroniczne rozwiązanie błędu dostępu w partycji głównej i wznowienie procesora wirtualnego po zakończeniu błędu dostępu. Kiedy mechanizm obsługi błędów stron „oświeconych” EPF (ang. *enlightened page fault*) jest włączony i następuje przechwycenie pamięci dla strony GPA wspieranej przez VA, sterownik VID w partycji głównej tworzy wątek roboczy w tle, który wywołuje obsługę błędu strony MicroVM i dostarcza synchroniczny wyjątek (nie mylić z asynchronicznym przerwaniem) do VP gościa w celu poinformowania go, że bieżący wątek spowodował przechwycenie pamięci.

Gość zmienia harmonogram wątku; w międzyczasie host obsługuje błąd dostępu. Gdy błąd dostępu zostanie zakończony, sterownik VID doda oryginalny błąd GPA do kolejki ukończenia i dostarczy asynchroniczne przerwanie do gościa. Przerwanie powoduje, że gość sprawdza kolejkę zakończenia i odblokowuje wszystkie wątki, które czekały na zakończenie EPF.

Odroczone zadeklarowanie i inne optymalizacje

Odroczone zadeklarowanie (ang. *deferred commit*) jest optymalizacją, która — jeżeli jest włączona — wymusza na sterowniku VID, aby każda strona rezerwowa nie była przydzielana aż do pierwszego dostępu. Dzięki temu możliwe jest jednoczesne działanie większej liczby maszyn wirtualnych bez zwiększania rozmiaru pliku stron. Jednak ze względu na to, że przestrzeń wspierająca wirtualne adresowanie jest tylko rezerwowana, a nie przydzielana, maszyny wirtualne mogą ulec awarii w czasie pracy z powodu osiągnięcia limitu zadeklarowania w partycji głównej. W takim przypadku nie ma już dostępnej wolnej pamięci.

Inne optymalizacje umożliwiają ustawienie wielkości stron, które będą przydzielone przez obsługę błędów stron wirtualnej mikromaszyny MicroVM (małe kontra duże) oraz *przypięcie* stron wspierających podczas pierwszego dostępu. Zapobiega to starzeniu się i przycinaniu, co ogólnie skutkuje bardziej spójną wydajnością, ale zużywa więcej pamięci i zmniejsza gęstość pamięci.

Proces VMMEM

Proces Pamięci maszyny wirtualnej VMMEM istnieje głównie z dwóch powodów:

- Hostuje pętlę wątków VP-dispatch, gdy włączony jest główny planista, który odzwierciedla podlegającą planowaniu jednostkę procesora wirtualnego gościa.
- Obsługuje przestrzeń adresów wirtualnych dla maszyn wirtualnych wspieranych przez VA.

Proces VMMEM jest tworzony przez sterownik VID podczas tworzenia partycji maszyny wirtualnej. Podobnie jak w przypadku zwykłych partycji (szczegóły w poprzednim punkcie), proces roboczy maszyny wirtualnej inicjalizuje konfigurację maszyny wirtualnej poprzez bibliotekę *VID.dll*, która wywołuje VID poprzez IOCTL. Jeśli sterownik VID wykryje, że nowa partycja jest wspierana przez VA, wywołuje wirtualną mikromaszynę (poprzez funkcję *VsmmNtSlatMemoryProcessCreate*) w celu utworzenia minimalnego procesu. MicroVM używa funkcji *PsCreateMinimalProcess*, która przydziela proces, tworzy jego przestrzeń adresową i umieszcza proces na liście procesów. Następnie rezerwuje dolne 4 GB przestrzeni adresowej, aby zapewnić, że nie trafią tam żadne bezpośrednio zmapowane obrazy (może to zmniejszyć entropię i bezpieczeństwo gościa). Sterownik VID nadaje nowemu procesowi VMMEM specjalny deskryptor bezpieczeństwa, do którego dostęp ma tylko SYSTEM i proces roboczy maszyny wirtualnej (proces roboczy maszyny wirtualnej jest uruchamiany z określonym tokenem; właściciel tokena jest ustawiony na identyfikator bezpieczeństwa SID wygenerowany z unikalnego GUID maszyny wirtualnej). Jest to ważne, ponieważ w przeciwnym razie wirtualna przestrzeń adresowa procesu VMMEM mogłaby być dostępna dla każdego. Czytając pamięć wirtualną procesu, złośliwy użytkownik mógłby odczytać prywatną pamięć fizyczną gościa VM.

Bezpieczeństwo oparte na wirtualizacji (VBS)

Jak omówiono w poprzednim punkcie, Hyper-V dostarcza usługi potrzebne do zarządzania i uruchamiania maszyn wirtualnych w systemach Windows. Hipernadzorca gwarantuje niezbędną izolację pomiędzy każdą partycją. W ten sposób maszyna wirtualna nie może zakłócać wykonywania innej. W tym rozdziale opisujemy kolejny ważny składnik infrastruktury wirtualizacji Windows: bezpieczne jądro, który dostarcza podstawowe usługi dla bezpieczeństwa opartego na wirtualizacji.

Najpierw wymieniamy usługi świadczone przez bezpieczne jądro i jego wymagania, a następnie opisujemy jego architekturę i podstawowe komponenty. Ponadto przedstawiamy niektóre z jego podstawowych wewnętrznych struktur danych. Następnie omawiamy metodę uruchamiania bezpiecznego jądra i wirtualnego trybu bezpiecznego (ang. *Virtual Secure Mode*), opisując jej dużą zależność od hipernadzorcy. Na koniec analizujemy komponenty, które są zbudowane na warstwie bezpiecznego jądra, takie jak tryb izolowanego użytkownika, integralność kodu wymuszona przez hipernadzorcę, bezpieczne enklawy programowe, bezpieczne urządzenia, oraz usługi takie jak jądro Windows łatane „na gorąco” i mikrokod.

Wirtualne poziomy zaufania (VTL) i wirtualny tryb bezpieczny (VSM)

Jak omówiono w poprzednim rozdziale, hipernadzorca wykorzystuje SLAT do utrzymania każdej partycji w jej własnej przestrzeni pamięci. System operacyjny, który działa w partycji, uzyskuje dostęp do pamięci w standardowy sposób (adresy wirtualne gościa są tłumaczone na adresy fizyczne gościa

za pomocą tablic stron). Pod tą osłoną sprzęt tłumaczy wszystkie GPA partycji na rzeczywiste SPA, a następnie wykonuje faktyczny dostęp do pamięci. Ta ostatnia warstwa translacji jest utrzymywana przez hipernadzorcę, który używa oddzielnej tablicy SLAT dla każdej partycji. W podobny sposób hipernadzorca może używać SLAT do tworzenia różnych domen bezpieczeństwa w jednej partycji. Dzięki tej właściwości Microsoft zaprojektował bezpieczne jądro, które jest podstawą Wirtualnego trybu bezpiecznego VSM (ang. *Virtual Secure Mode*).

Tradycyjnie system operacyjny miał jedną fizyczną przestrzeń adresową, a oprogramowanie uruchomione w pierścieniu 0 (czyli w trybie jądra) mogło mieć dostęp do dowolnego adresu pamięci fizycznej. Jeśli więc bezpieczeństwo jakiegokolwiek oprogramowania działającego w trybie nadzorcy (jądro, sterowniki i tak dalej) zostanie naruszone, bezpieczeństwo całego systemu również zostanie naruszone. Wirtualny tryb bezpieczny wykorzystuje hipernadzorcę do zapewnienia nowych granic zaufania dla oprogramowania systemowego. Dzięki VSM można wprowadzić granice bezpieczeństwa (opisane przez hipernadzorcę za pomocą SLAT), które ograniczają zasoby, do których może mieć dostęp kod trybu nadzorcy. W ten sposób, dzięki VSM, nawet jeśli kod trybu nadzorcy jest zagrożony, cały system nie jest zagrożony.

VSM zapewnia te granice dzięki koncepcji wirtualnych poziomów zaufania (VTL — ang. *Virtual Trust Level*). W swojej istocie VTL to zestaw zabezpieczeń dostępu do pamięci fizycznej. Każdy VTL może mieć inny zestaw zabezpieczeń dostępu. W ten sposób VTL-e mogą być wykorzystane do zapewnienia izolacji pamięci. Zabezpieczenia dostępu do pamięci VTL-a mogą być skonfigurowane tak, aby ograniczyć dostęp do pamięci fizycznej, do której VTL może uzyskać dostęp. Dzięki trybowi VSM wirtualny procesor jest zawsze uruchomiony w konkretnym VTL-u i może uzyskać dostęp tylko do pamięci fizycznej, która jest oznaczona jako dostępna przez SLAT hipernadzorcy. Na przykład, jeśli procesor działa w VTL 0, może uzyskać dostęp tylko do pamięci kontrolowanej przez zabezpieczenia dostępu do pamięci związane z VTL 0. To egzekwowanie dostępu do pamięci odbywa się na poziomie translacji pamięci fizycznej gościa i dlatego nie może być zmienione przez kod trybu nadzorcy w partycji.

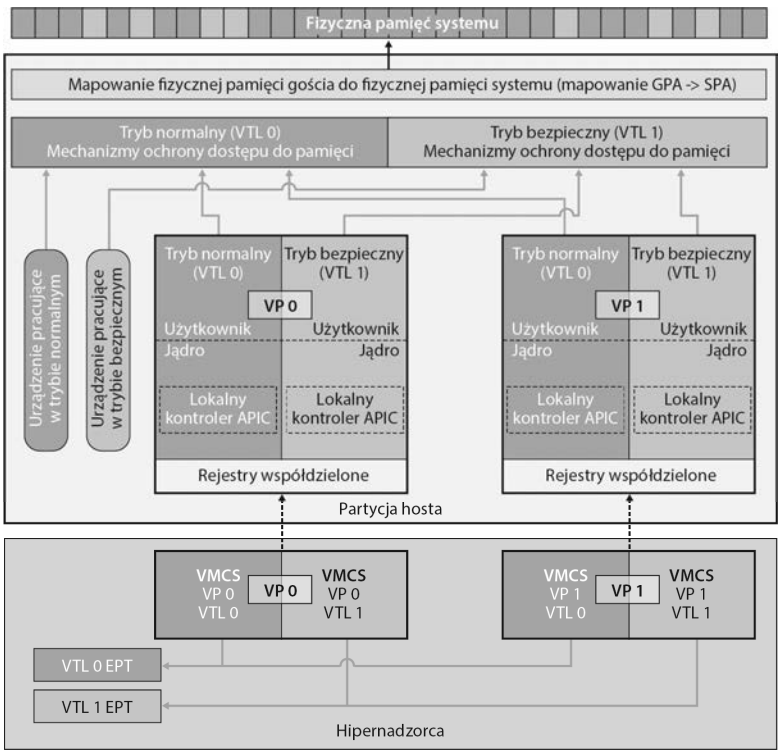
VTL-e są zorganizowane w sposób hierarchiczny. Wyższe poziomy są bardziej uprzywilejowane niż niższe i mogą regulować zabezpieczenia dostępu do pamięci dla niższych poziomów. Tak więc oprogramowanie działające na poziomie VTL 1 może dostosować zabezpieczenia dostępu do pamięci VTL 0, aby ograniczyć to, do czego może mieć dostęp pamięć VTL 0. Pozwala to oprogramowaniu w VTL 1 ukryć (odizolować) pamięć od VTL 0. Jest to ważna koncepcja, która jest podstawą trybu VSM. Obecnie hipernadzorca obsługuje tylko dwa VTL-e: VTL 0 odzwierciedla środowisko wykonawcze normalnego systemu operacyjnego, z którym użytkownik wchodzi w interakcję; VTL 1 odzwierciedla tryb bezpieczny (ang. *Secure Mode*), gdzie działają bezpieczne jądro (ang. *Secure Kernel*) i tryb izolowanego użytkownika (IUM — ang. *Isolated User Mode*). Ponieważ VTL 0 jest środowiskiem, w którym działa standardowy system operacyjny i aplikacje, często określany jest jako tryb normalny.

Uwaga. Architektura VSM została początkowo zaprojektowana do obsługi maksymalnie 16 VTL-ów. W chwili pisania tego tekstu tylko 2 VTL-e są obsługiwane przez hipernadzorcę. W przyszłości może się zdarzyć, że Microsoft doda jeden lub większą ilość nowych VTL-ów. Na przykład najnowsze wersje systemu Windows Server działające w Azure obsługują również maszyny wirtualne Confidential, które uruchamiają swoją warstwę kompatybilności hosta (HCL — ang. *Host Compatibility Layer*) w VTL 2.

Z każdym VTL-em związane są następujące cechy:

- **Ochrona dostępu do pamięci** — Jak już wspomniano, każdy wirtualny poziom zaufania posiada zestaw zabezpieczeń dostępu do pamięci fizycznej gościa, który definiuje, w jaki sposób oprogramowanie może uzyskać dostęp do pamięci.
- **Stan procesora wirtualnego** — Procesor wirtualny w hipernadzorcy współdzieli niektóre rejestry z każdym VTL-em, natomiast niektóre inne rejestry są prywatne dla każdego VTL-a. Prywatny stan wirtualnego procesora dla VTL-a nie może być dostępny dla oprogramowania działającego w niższym VTL-u. Pozwala to na izolację stanu procesora pomiędzy VTL-ami.
- **Podsystem przerwań** — Każdy VTL posiada unikalny podsystem przerwań (zarządzany przez Syntetyczny kontroler przerwań (SynIC) hipernadzorcy). Podsystem przerwań VTL-a nie może być dostępny dla oprogramowania działającego w niższym VTL-u. Pozwala to na bezpieczne zarządzanie przerwaniami w danym VTL-u bez ryzyka, że niższy VTL wygeneruje nieoczekiwane przerwania lub zamaskuje przerwania.

Rysunek 9.30 przedstawia schemat ochrony pamięci zapewnianej przez hipernadzorcę trybowi VSM (ang. *Virtual Secure Mode*). Hipernadzorca odzwierciedla każdy VTL wirtualnego procesora poprzez inną strukturę danych VMCS (więcej szczegółów w poprzednim punkcie), która zawiera specyficzną tablicę SLAT. W ten sposób oprogramowanie, które działa w konkretnym VTL-u, ma dostęp tylko do stron pamięci fizycznej przypisanych do jego poziomu. Ważną koncepcją jest to, że ochrona SLAT jest stosowana do stron *fizycznych*, a nie do stron *wirtualnych*, które są chronione przez standardowe tablice stron.



RYSUNEK 9.30. *Schemat architektury ochrony pamięci dostarczanej przez hipernadzorcę do wirtualnego trybu bezpiecznego (VSM)*

Usługi świadczone przez VSM i wymagania

Wirtualny tryb bezpieczny (VSM), który jest zbudowany na warstwie hipernadzorcy, dostarcza następujące usługi do ekosystemu Windows:

- **Izolacja** — Tryb izolowanego użytkownika IUM (ang. *Isolated User Mode*) zapewnia sprzętowo izolowane środowisko dla każdego oprogramowania, które działa w VTL 1. Bezpieczne urządzenia zarządzane przez bezpieczne jądro są odizolowane od reszty systemu i działają w trybie użytkownika VTL 1. Oprogramowanie działające w VTL 1 zazwyczaj przechowuje tajne dane, które nie mogą być przechwycone lub ujawnione w VTL 0. Usługa ta jest intensywnie wykorzystywana przez strażnika poświadczeń (ang. *Credential Guard*). Strażnik poświadczeń to funkcja, która przechowuje wszystkie poświadczenia systemowe w przestrzeni adresowej pamięci trustletu LsaIso, działającego w trybie użytkownika VTL 1.
- **Kontrola nad VTL 0** — Integralność kodu wymuszona przez hipernadzorcę HVCI (ang. *Hypervisor Enforced Code Integrity*) sprawdza integralność i podpisanie każdego modułu ładowanego i uruchamianego przez normalny system operacyjny. Sprawdzanie integralności odbywa się w całości w VTL 1 (który ma dostęp do całej pamięci fizycznej VTL 0). Żadne oprogramowanie VTL 0 nie może ingerować w kontrolę podpisywania. Ponadto HVCI gwarantuje, że wszystkie strony pamięci trybu normalnego, które zawierają kod wykonywalny, są oznaczone jako niezapisywalne (ta cecha nazywa się W^X. Zarówno HVCI, jak i W^X zostały omówione w rozdziale 7. części I).
- **Bezpieczne przechwyty** — VSM zapewnia mechanizm pozwalający wyższemu VTL-owi na zablokowanie krytycznych zasobów systemowych i uniemożliwienie dostępu do nich niższym VTL-om. Bezpieczne przechwyty są szeroko wykorzystywane przez hiperstrażnika (ang. *HyperGuard*), który zapewnia kolejną warstwę ochrony dla jądra VTL 0, zatrzymując złośliwe modyfikacje krytycznych komponentów systemów operacyjnych.
- **Enklawy oparte na VBS** — Enklawa bezpieczeństwa to izolowany region pamięci w przestrzeni adresowej procesu pracującego w trybie użytkownika. Region pamięci enklawy jest niedostępny nawet dla wyższych poziomów uprawnień. Oryginalna implementacja tej technologii wykorzystywała sprzętowe udogodnienia do odpowiedniego szyfrowania pamięci należącej do procesu. Enklawa oparta na VBS jest bezpieczną enklawą, której gwarancje izolacji są zapewnione przy użyciu VSM.
- **Ochrona przepływu sterowania jądra KCFG** (ang. *Kernel Control Flow Guard*) — VSM, gdy HVCI jest włączone, zapewnia ochronę przepływu sterowania CFG (ang. *Control Flow Guard*) każdemu modułowi jądra załadowanego w normalnym świecie (i samemu jądru NT). Oprogramowanie w trybie jądra pracujące w normalnym świecie ma dostęp do mapy bitowej w trybie tylko do odczytu, więc exploit nie może potencjalnie zmodyfikować tej mapy. Z tego powodu CFG jądra w Windows jest również znane jako bezpieczne jądro CFG (SKCFG).

Uwaga. Ochrona przepływu sterowania CFG jest implementacją Integralności przepływu sterowania CFI (ang. Control Flow Integrity) — techniki autorstwa Microsoftu. Technika ta nie pozwala szerokiej gamie złośliwych ataków na przekierowywanie przepływu wykonania programu. Zarówno CFG w trybie użytkownika, jak i CFG w trybie jądra zostały szeroko omówione w rozdziale 7. części I.

- **Urządzenia bezpieczne** — Urządzenia bezpieczne to nowy rodzaj urządzeń, które są mapowane i zarządzane w całości przez bezpieczne jądro w VTL 1. Sterowniki dla tego rodzaju urządzeń pracują całkowicie w trybie użytkownika VTL 1 i używają usług dostarczanych przez bezpieczne jądro do mapowania przestrzeni we/wy urządzenia.

Aby VSM został prawidłowo włączony i działał poprawnie, istnieją pewne wymagania sprzętowe. System hosta musi obsługiwać rozszerzenia wirtualizacji (VT-x Intela, SVM firmy AMD lub strefa zaufania (TrustZone) firmy ARM) oraz SLAT. VSM nie będzie działać, jeśli jedna z poprzednich cech sprzętowych nie jest obecna w procesorze systemowym. Niektóre inne cechy sprzętowe nie są bezwzględnie konieczne, ale w przypadku ich braku niektóre założenia bezpieczeństwa VSM mogą nie być zagwarantowane:

- IOMMU jest potrzebny do ochrony przed atakami DMA na urządzenia fizyczne. Jeśli procesory systemowe nie mają IOMMU, VSM może nadal działać, ale jest podatny na ataki na urządzenia fizyczne.
- UEFI BIOS z włączonym bezpiecznym rozruchem jest potrzebny do ochrony łańcucha rozruchowego, który prowadzi do uruchomienia hipernadzorcy i bezpiecznego jądra. Jeśli bezpieczne jądro nie jest włączone, system jest podatny na ataki startowe, które mogą zmodyfikować integralność hipernadzorcy i bezpieczne jądro, zanim będą miały szansę na wykonanie swoich instrukcji.

Niektóre inne komponenty są opcjonalne, ale gdy są obecne, zwiększają ogólne bezpieczeństwo i szybkość reakcji systemu. Dobrym przykładem jest obecność modułu zaufanej platformy (TPM — ang. *Trusted Platform Module*). Jest on wykorzystywany przez bezpieczne jądro do przechowywania głównego klucza szyfrowania oraz do wykonywania bezpiecznego uruchomienia (znanego również jako dynamiczny root pomiaru zaufania (DRTM — ang. *Dynamic Root of Trust Measurement*); więcej szczegółów w rozdziale 12.). Innym składnikiem sprzętowym, który może poprawić reaktywność trybu VSM, jest sprzętowa obsługa mechanizmu kontroli pracy procesora zależna od trybu działania procesora (MBEC — ang. *Mode-Based Execute Control*). MBEC jest używane, gdy włączone jest HVCI, aby chronić stan wykonania stron trybu użytkownika w trybie jądra. Dzięki sprzętowemu MBEC, hipernadzorca może ustawić stan wykonawczy strony pamięci fizycznej w oparciu o domenę CPL (jądra lub użytkownika) konkretnego VTL-a. W ten sposób pamięć należąca do aplikacji trybu użytkownika może być fizycznie oznaczona jako wykonywalna tylko przez kod trybu użytkownika (exploity jądra nie mogą już wykonywać własnego kodu znajdującego się w pamięci aplikacji trybu użytkownika). W przypadku braku sprzętowego MBEC hipernadzorca musi go emulować, używając dwóch różnych tablic SLAT dla VTL 0 i przełączając je, gdy wykonanie kodu zmienia domenę bezpieczeństwa CPL (przejście z trybu użytkownika do trybu jądra i odwrotnie powoduje w tym przypadku VMEXIT). Więcej szczegółów na temat HVCI zostało już omówionych w rozdziale 7. części I.

Eksperyment: wykrywanie VBS i jego usług

W rozdziale 12. omawiamy politykę uruchamiania VSM i przedstawiamy instrukcje ręcznego włączania lub wyłączania bezpieczeństwa opartego na wirtualizacji VBS (ang. *Virtualization-Based Security*). W tym eksperymencie określamy stan różnych funkcji dostarczanych przez hipernadzorcę i bezpieczne jądro. VBS jest technologią, która nie jest bezpośrednio widoczna dla użytkownika. Narzędzie informacji o systemie (ang. *System Information*) dystrybuowane z bazową instalacją Windows jest w stanie pokazać szczegóły dotyczące bezpiecznego jądra i powiązanych z nim technologii. Można je uruchomić, wpisując *msinfo32* w polu wyszukiwania Cortany. Pamiętaj, aby uruchomić to narzędzie z poziomu konta Administratora; niektóre szczegóły wymagają konta użytkownika z pełnymi uprawnieniami.

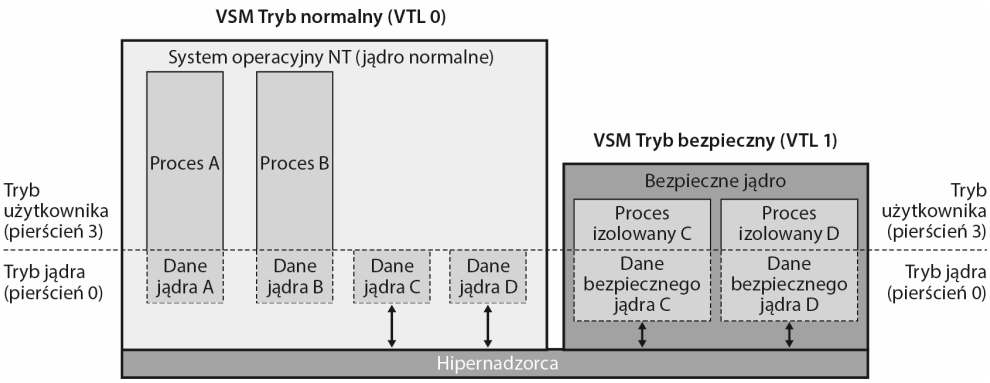
Na poniższym rysunku VBS jest włączony i zawiera HVCI (określone jako *Hypervisor Enforced Code Integrity*), wirtualizację wykonawczą UEFI (określoną jako *UEFI Code Readonly*), MBEC (określone jako *Mode Based Execution Control*). Jednakże system opisany w tym przykładzie nie zawiera włączonego bezpiecznego rozruchu i nie posiada działającego IOMMU (określonego jako ochrona DMA w linii właściwości zabezpieczeń dostępnych w oparciu o bezpieczną wirtualizację).

Więcej szczegółów na temat włączania, wyłączania i blokowania konfiguracji VBS jest dostępnych w eksperymencie „Badanie zasady trybu VSM" w rozdziale 12.

Bezpieczne jądro

Bezpieczne jądro jest zaimplementowane głównie w pliku *securekernel.exe* i jest uruchamiane przez Windows Loader po wcześniejszym udanym uruchomieniu hipernadzorcy. Jak pokazano na rysunku 9.31, bezpieczne jądro to minimalny system operacyjny, który ściśle współpracuje z normalnym jądrem, rezydującym w VTL 0. Jak każdy normalny system operacyjny bezpieczne jądro działa na poziomie przywileju kodu (CPL — ang. *Code Privilege Level*) równym 0 (znanym również jako pierścień 0 lub tryb jądra) dla VTL 1 i dostarcza usługi (większość z nich poprzez wywołania systemowe) do trybu izolowanego użytkownika IUM (ang. *Isolated User Mode*), który żyje w CPL 3 (znanym również jako pierścień 3 lub tryb użytkownika) dla VTL 1. Bezpieczne jądro zostało zaprojektowane tak, aby było tak małe jak to tylko możliwe, a jego celem jest zmniejszenie zewnętrznej powierzchni narażonej na atak. Nie jest rozszerzalne o zewnętrzne sterowniki urządzeń jak zwykłe jądro. Jedyne moduły jądra, które rozszerzają jego funkcjonalność, są ładowane przez Windows Loader przed uruchomieniem wirtualnego trybu bezpieczeństwa VSM i są importowane z *securekernel.exe*:

- *Skci.dll* — implementuje część integralności kodu wymuszonej przez hipernadzorcę HVCI należącą do bezpiecznego jądra;
- *Cng.sys* — zapewnia silnik kryptograficzny dla bezpiecznego jądra;
- *Vmsvcext.dll* — zapewnia wsparcie dla atestacji komponentów bezpiecznego jądra w środowiskach Intel TXT (zaufany rozruch (ang. *Trusted Boot*) — więcej informacji o zaufanym rozruchu można znaleźć w rozdziale 12.).



RYSUNEK 9.31. *Schemat architektury wirtualnego trybu bezpiecznego, zbudowanego na warstwie hipernadzorcy*

Podczas gdy bezpieczne jądro nie jest rozszerzalne, tryb izolowanego użytkownika IUM zawiera wyspecjalizowane procesy zwane *trustletami*. Trustlety są izolowane między sobą i mają wyspecjalizowane wymagania dotyczące podpisu cyfrowego. Mogą one komunikować się z bezpiecznym jądrem poprzez wywołania systemowe (syscalls) i z normalnym światem poprzez Mailslots i ALPC. Tryb izolowanego użytkownika IUM jest omówiony w dalszej części tego rozdziału.

Wirtualne przerwania

Kiedy hipernadzorca konfiguruje bazowe partycje wirtualne, wymaga od fizycznych procesorów wytworzenia VMEXIT za każdym razem, gdy zewnętrzne przerwanie zostanie podniesione przez fizyczny zaawansowany programowalny kontroler przerwań APIC procesora. Sprzętowe rozszerzenia maszyny wirtualnej pozwalają hipernadzorcy wstrzykiwać wirtualne przerwania do partycji gościa (więcej szczegółów w podręcznikach użytkownika Intel, AMD i ARM). Dzięki tym dwóm faktom hipernadzorca implementuje pojęcie syntetycznego kontrolera przerwań (SynIC). SynIC może zarządzać dwoma rodzajami przerwań. Przerwania wirtualne to przerwania dostarczane do wirtualnego APIC partycji gościa. Przerwanie wirtualne może odzwierciedlać fizyczne przerwanie sprzętowe, które jest generowane przez rzeczywisty sprzęt, i może być z nim powiązane. W przeciwnym razie przerwanie wirtualne może odzwierciedlać przerwanie syntetyczne, które jest generowane przez samego hipernadzorcę w odpowiedzi na pewne rodzaje zdarzeń. SynIC może mapować fizyczne przerwania do tych wirtualnych. VTL ma SynIC przypisany do każdego wirtualnego procesora, w którym działa VTL. Aktualnie (w chwili powstawania tego tekstu) hipernadzorca jest zaprojektowany do obsługi 16 różnych wektorów przerwań syntetycznych (w rzeczywistości jednak tylko 2 są w użyciu).

Podczas startu systemu (faza pierwsza inicjalizacji jądra NT) sterownik ACPI mapuje każde przerwanie do właściwego wektora, korzystając z usług dostarczanych przez HAL. Warstwa abstrakcji sprzętu HAL jądra NT jest „oświecona" i wie, czy działa pod VSM. W takim przypadku wysyła wywołanie do hipernadzorcy w celu mapowania każdego fizycznego przerwania do jego własnego VTL-a. Nawet bezpieczne jądro mogłoby zrobić to samo. W momencie pisania tego tekstu, żadne fizyczne przerwania nie są związane z bezpiecznym jądrem (może się to zmienić w przyszłości; hipernadzorca już obsługuje tę funkcję). Bezpieczne jądro prosi hipernadzorcę o odbieranie tylko następujących przerwań wirtualnych: Secure Timers, Virtual Interrupt Notification Assist (VINA) oraz Secure Intercepts.

Uwaga. Ważne jest, aby zrozumieć, że hipernadzorca wymaga, aby sprzęt bazowy wytworzył VMEXIT podczas zarządzania przerwaniami, które są tylko typu zewnętrznego. Wyjątki są nadal zarządzane w tym samym VTL-u, w którym wykonuje instrukcje procesor (nie jest generowany VMEXIT). Jeśli instrukcja powoduje wyjątek, jest on nadal zarządzany przez kod strukturalnej obsługi wyjątków SEH (ang. *structured exception handling*) znajdujący się w bieżącym VTL-u.

Aby zrozumieć trzy rodzaje przerwań wirtualnych, musimy najpierw opisać, w jaki sposób przerwania są zarządzane przez hipernadzorcę.

W hipernadzorcy każdy VTL został zaprojektowany tak, aby bezpiecznie odbierał przerwania od urządzeń związanych z jego własnym VTL-em, po to aby posiadał bezpieczny obiekt czasowy, w który nie mogą ingerować mniej bezpieczne VTL-e, oraz aby mógł zapobiegać przerwaniom kierowanym do niższych VTL-ów podczas wykonywania kodu w wyższym VTL-u. Ponadto VTL powinien móc wysyłać przerwania IPI do innych procesorów. Taki projekt systemu daje następujące scenariusze:

- Podczas pracy w danym VTL-u odbiór przerwań skierowanych do aktualnego VTL-a powoduje standardową obsługę przerwań (określoną przez wirtualny kontroler APIC procesora wirtualnego).
- Gdy odbierane jest przerwanie skierowane do wyższego VTL-a, odbiór przerwania powoduje przełączenie do wyższego VTL-a, do którego skierowane jest przerwanie, jeśli wartość IRQL dla wyższego VTL-a pozwoliłaby na prezentację przerwania.

Jeśli wartość IRQL wyższego VTL-a nie pozwala na dostarczenie przerwania, przerwanie jest umieszczane w kolejce bez przełączania bieżącego VTL-a. Takie zachowanie pozwala wyższemu VTL-owi selektywnie maskować przerwania podczas powrotu do niższego VTL-a. Może to być przydatne, gdy wyższy VTL wykonuje procedurę obsługi przerwania i musi wrócić do niższego VTL-a po pomoc w przetworzeniu przerwania.

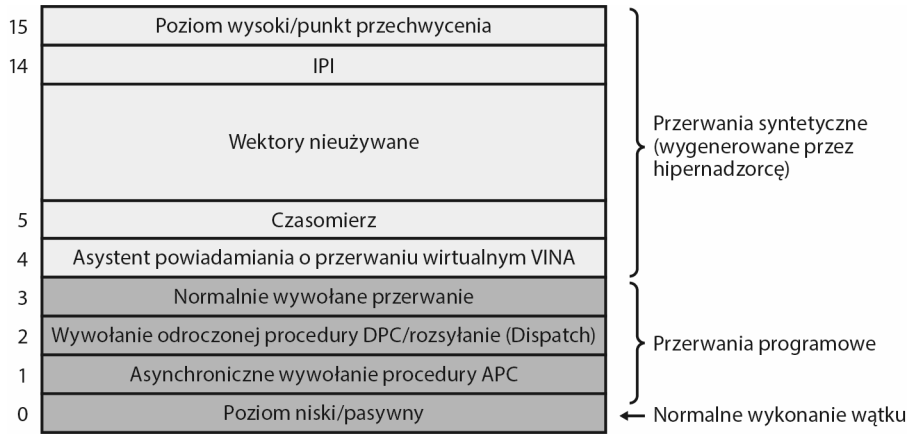
- Gdy odbierane jest przerwanie skierowane do VTL-a niższego niż aktualnie wykonujący instrukcje VTL procesora wirtualnego, przerwanie jest umieszczane w kolejce dla przyszłego dostarczenia do niższego VTL-a. Przerwanie skierowane do niższego VTL-a nigdy nie spowoduje wyłączenia wykonywania bieżącego VTL-a. Zamiast tego przerwanie jest prezentowane podczas kolejnej transformacji wirtualnego procesora do docelowego VTL-a.

Zapobieganie przerwaniom skierowanym do niższych VTL-ów nie zawsze jest dobrym rozwiązaniem. W wielu przypadkach *może* to prowadzić do spowolnienia normalnego wykonywania systemu operacyjnego (zwłaszcza w środowiskach krytycznych lub grach). Aby lepiej zarządzać tymi warunkami, wprowadzono VINA. Jako część swojej normalnej pętli wysyłania zdarzeń hipernadzorca sprawdza, czy istnieją oczekujące przerwania ustawione w kolejce do niższego VTL-a. Jeśli tak, hipernadzorca wstrzykuje przerwanie VINA do aktualnie wykonywanego VTL-a. Bezpieczne jądro posiada handler zarejestrowany dla wektora VINA w jego wirtualnej tablicy deskryptorów przerwań IDT. Handler (funkcja `ShvlVinaHandler`) wykonuje normalne wywołanie (`NORMALKERNEL_VINA`) do VTL 0 (wywołania normalne i bezpieczne omówiono w dalszej części rozdziału). To wywołanie zmusza hipernadzorcę do przełączenia się na normalne jądro (VTL 0). Tak długo jak VTL jest przełączone, wszystkie przerwania w kolejce zostaną prawidłowo rozesłane. Normalne jądro ponownie wejdzie do VTL 1, emitując bezpieczne wywołanie `SECUREKERNEL_RESUMETHREAD`.

Bezpieczne poziomy żądań przerwania IRQL

Handler VINA nie zawsze będzie wykonywany w VTL 1. Podobnie jak w jądrze NT, zależy to od aktualnego IRQL, w którym jest wykonywany kod. Poziom żądania przerwania IRQL aktualnie wykonywanego kodu maskuje wszystkie przerwania, które są powiązane z poziomem IRQL mniejszym lub równym wyżej wymienionemu poziomowi. Mapowanie pomiędzy wektorem przerwań a IRQL jest utrzymywane przez rejestr priorytetów zadań TPR (ang. *task priority register*) wirtualnego kontrolera APIC, podobnie jak w przypadku prawdziwych fizycznych kontrolerów APIC (więcej informacji na ten temat można znaleźć w podręczniku Intel Architecture Manual). Jak widać na rysunku 9.32, bezpieczne jądro obsługuje różne poziomy IRQL w porównaniu z normalnym jądrem. Te IRQL są nazywane bezpiecznymi (ang. *secure*) IRQL.

Pierwsze trzy poziomy żądania przerwań bezpiecznego jądra IRQL są zarządzane przez bezpieczne jądro w sposób podobny do normalnego świata. Normalne APC i DPC (skierowane do VTL 0) nadal nie mogą uprzedzić kodu wykonywanego w VTL 1 poprzez hipernadzorcę, ale przerwanie VINA jest wciąż dostarczane do bezpiecznego jądra (system operacyjny zarządza trzema przerwaniami programowymi poprzez zapis w rejestrze priorytetów zadań APIC procesora docelowego. Jest to operacja, która powoduje VMEXIT do hipernadzorcy. Więcej informacji na temat rejestru TPR kontrolera APIC można znaleźć w podręcznikach firm Intel, AMD lub ARM). Oznacza to, że jeśli DPC w trybie normalnym zostanie skierowane do procesora podczas wykonywania kodu VTL 1 (przy zgodnym bezpiecznym IRQL, który powinien być mniejszy niż rozsyłanie (*Dispatch*)), to przerwanie VINA zostanie dostarczone i przełączy kontekst wykonania na VTL 0. W gruncie rzeczy powoduje to wykonanie DPC w normalnym świecie i podnosi na chwilę IRQL normalnego jądra do poziomu rozsyłania (ang. *dispatch*).



RYSUNEK 9.32. *Poziomy żądania przerwań bezpiecznego jądra (IRQL)*

Kiedy kolejka DPC zostaje opróżniona, poziom IRQL normalnego jądra obniża się. Przepływ wykonania wraca do bezpiecznego jądra dzięki kodowi pętli komunikacyjnej VSM, który znajduje się w procedurze *VslpEnterIumSecureMode*. Pętla ta przetwarza każde normalne wywołanie pochodzące z bezpiecznego jądra.

Bezpieczne jądro mapuje pierwsze trzy bezpieczne IRQL do tego samego IRQL normalnego świata. Gdy wywołanie bezpieczne jest realizowane z kodu wykonywanego na określonym poziomie IRQL (jeszcze mniejszym lub równym poziomowi rozsyłania) w normalnym świecie, bezpieczne jądro przełącza swój własny bezpieczny IRQL na ten sam poziom. I odwrotnie, gdy bezpieczne jądro wykonuje normalne wywołanie, aby wejść do jądra NT, przełącza IRQL normalnego jądra na ten sam poziom co jego własny. Działa to tylko dla pierwszych trzech poziomów.

Normalny podniesiony poziom jest używany, gdy jądro NT wchodzi do bezpiecznego świata na IRQL wyższym niż poziom DPC. W takich przypadkach bezpieczne jądro mapuje wszystkie IRQL z normalnego świata, które są na poziomie powyżej DPC, do swojego normalnego podniesionego bezpiecznego poziomu. Kod bezpiecznego jądra wykonywany na tym poziomie nie może otrzymać VINA dla żadnego rodzaju programowych IRQL w normalnym jądrze (ale nadal może otrzymać VINA dla przerwań sprzętowych). Za każdym razem gdy jądro NT wchodzi w bezpieczny świat na normalnym poziomie IRQL powyżej poziomu DPC, bezpieczne jądro podnosi swój bezpieczny poziom IRQL do normalnego podniesionego.

Bezpieczne IRQL równe lub wyższe od VINA nigdy nie może być poprzedzone przez żaden kod w normalnym świecie. To wyjaśnia, dlaczego bezpieczne jądro obsługuje koncepcję bezpiecznych, nieprzewidywalnych czasomierzy i bezpiecznych przechwytów. Bezpieczne czasomierze są generowane z poziomu procedury obsługi przerwania ISR zegara hipernadzorcy. Ta procedura ISR przed wstrzyknięciem syntetycznego przerwania zegara do jądra NT sprawdza, czy istnieje jeden lub większa liczba bezpiecznych czasomierzy, które wygasły. Jeśli tak, to wstrzykuje syntetyczne bezpieczne przerwanie zegarowe do VTL 1. Następnie przystępuje do przekazania przerwania zegarowego do normalnego VTL-a.

Bezpieczne przechwyty

Istnieją przypadki, w których bezpieczne jądro może musieć uniemożliwić jądru NT wydającemu instrukcje na niższym VTL-u dostęp do pewnych krytycznych zasobów systemowych. Na przykład zapisy w MSR-ach niektórych procesorów mogą być potencjalnie wykorzystane do przeprowadzenia ataku, który wyłączy hipernadzorcę lub pokona niektóre z jego zabezpieczeń. Wirtualny tryb bezpieczny VSM dostarcza mechanizm pozwalający wyższym VTL-om zablokować krytyczne zasoby systemowe i uniemożliwić dostęp do nich niższym VTL-om. Mechanizm ten nazywany jest *bezpiecznymi przechwytami*.

Bezpieczne przechwyty są implementowane w bezpiecznym jądrze poprzez rejestrację syntetycznego przerwania, które jest dostarczane przez hipernadzorcę (ponownie mapowane w bezpiecznym jądrze do wektora 0xF0). Gdy pewne zdarzenia powodują VMEXIT, hipernadzorca wstrzykuje syntetyczne przerwanie do wyższego VTL-a na wirtualnym procesorze, który wywołał przechwycenie. W czasie pisania tego tekstu, bezpieczne jądro może rejestrować się u hipernadzorcy dla następujących typów przechwytywanych zdarzeń:

- zapis do niektórych istotnych MSR-ów procesora (Star, Lstar, Cstar, Efer, Sysenter, Ia32Misc i APIC base w architekturach AMD64) i rejestrów specjalnych (GDT, IDT, LDT);
- zapis do niektórych rejestrów kontrolnych (CR0, CR4 i XCR0);
- zapis do niektórych portów we/wy (porty 0xCF8 i 0xCFC są dobrymi przykładami; przechwytywanie zarządza ponowną konfiguracją urządzeń PCI);
- nieprawidłowy dostęp do chronionej pamięci fizycznej gościa.

Kiedy oprogramowanie VTL 0 powoduje przechwycenie, którego poziom zostanie podniesiony w VTL 1, bezpieczne jądro musi rozpoznać typ przechwycenia ze swojej procedury obsługi przerwania. W tym celu bezpieczne jądro wykorzystuje kolejkę komunikatów przydzieloną przez SynIC dla źródła przerwania syntetycznego „przechwytu" (więcej szczegółów na temat SynIC i SINT znajduje się w punkcie „Komunikacja między partycjami" we wcześniejszej części tego rozdziału). Bezpieczne jądro jest w stanie odkryć i zmapować stronę pamięci fizycznej poprzez sprawdzenie syntetycznego MSR-a SIMP, który jest zwirtualizowany przez hipernadzorcę. Mapowanie strony fizycznej jest wykonywane w czasie inicjalizacji bezpiecznego jądra w VTL 1. Uruchomienie bezpiecznego jądra jest opisane w dalszej części tego rozdziału.

Przechwyty są szeroko stosowane przez mechanizm HyperGuard w celu ochrony wrażliwych części normalnego jądra NT. Jeżeli złośliwy rootkit zainstalowany w jądrze NT próbuje zmodyfikować system poprzez zapisanie określonej wartości w chronionym rejestrze (na przykład w rejestrach syscall handlers, CSTAR i LSTAR lub w rejestrach specyficznych dla danego modelu), handler bezpiecznego jądra (*ShvlpInterceptHandler*) filtruje wartość nowego rejestru, a jeżeli odkryje, że wartość nie jest akceptowalna, wstrzykuje niemaskowalny wyjątek Błąd ochrony ogólnej GPF (ang. *General Protection Fault*) do jądra NT w VTL 0. Powoduje to natychmiastowe sprawdzenie buga (ang. *bugcheck*), w wyniku którego system zostaje zatrzymany. Jeśli wartość jest akceptowalna, bezpieczne jądro zapisuje nową wartość rejestru za pomocą hipernadzorcy poprzez hiperwywołanie *HvSetVpRegisters* (w tym przypadku bezpieczne jądro pośredniczy w dostępie do rejestru).

Kontrola nad hiperwywołaniami

Ostatnim typem przechwytu, który bezpieczne jądro rejestruje u hipernadzorcy, jest przechwyt typu hiperwywołanie. Handler przechwytujący hiperwywołanie sprawdza, czy hiperwywołanie emitowane przez kod VTL 0 do hipernadzorcy jest legalne i pochodzi z samego systemu operacyjnego, a nie przez jakieś zewnętrzne moduły. Za każdym razem, gdy w dowolnym VTL-u emitowane jest hiperwywołanie, powoduje ono VMEXIT u hipernadzorcy (z założenia). Hiperwywołania są podstawową usługą używaną przez komponenty jądra każdego VTL-a do żądania usług pomiędzy sobą (i do samego hipernadzorcy). Hipernadzorca wstrzykuje syntetyczne przerwanie przechwytujące do wyższego VTL-a tylko dla hiperwywołań używanych do żądania usług bezpośrednio do hipernadzorcy, pomijając wszystkie hiperwywołania używane do bezpiecznych i normalnych połączeń do i z bezpiecznego jądra.

Jeśli hiperwywołanie nie zostanie rozpoznane jako poprawne, nie zostanie wykonane. Bezpieczne jądro w tym przypadku aktualizuje rejestry niższego VTL-a w celu zasygnalizowania błędu hiperwywołania. System nie ulega awarii (choć to zachowanie może się zmienić w przyszłości); kod wywołujący może zdecydować, jak zarządzać błędem.

Wywołania systemowe wirtualnego trybu bezpiecznego VSM

Jak opisaliśmy to w poprzednich punktach, wirtualny tryb bezpieczny (VSM — ang. *Virtual Secure Mode*) używa hiperwywołań do żądania usług do i od bezpiecznego jądra. Hiperwywołania zostały pierwotnie zaprojektowane jako sposób żądania usług skierowany do hipernadzorcy, ale w VSM model ten został rozszerzony o obsługę nowych typów wywołań systemowych:

- Bezpieczne wywołania są emitowane przez normalne jądro NT w VTL 0 do bezpiecznego jądra, po to aby żądać usług.
- Normalne wywołania są żądane przez bezpieczne jądro w VTL 1, gdy potrzebuje ono usług świadczonych przez jądro NT, które działa w VTL 0. Ponadto niektóre z nich są wykorzystywane przez bezpieczne procesy (trustlety) działające w trybie izolowanego użytkownika (IUM — ang. *Isolated User Mode*) do żądania usług od bezpiecznego jądra lub normalnego jądra NT.

Tego typu wywołania systemowe są zaimplementowane u hipernadzorcy, w bezpiecznym jądrze i normalnym jądrze NT. Hipernadzorca definiuje dwa hiperwywołania służące do przełączania się między różnymi VTL-ami: *HvVtlCall* i *HvVtlReturn*. Bezpieczne jądro i jądro NT definiują pętlę rozsyłania używaną do wysyłania bezpiecznych i normalnych wywołań.

Ponadto bezpieczne jądro implementuje inny rodzaj wywołań systemowych: bezpieczne wywołania systemowe. Świadczą one usługi tylko dla bezpiecznych procesów (trustletów), które działają w trybie IUM. Te wywołania systemowe nie są prezentowane normalnemu jądru NT. Hipernadzorca nie jest w ogóle zaangażowany podczas przetwarzania bezpiecznych wywołań systemowych.

Stan procesora wirtualnego

Przed zagłębieniem się w architekturę wywołania bezpiecznego i normalnego przeanalizuj, w jaki sposób wirtualny procesor zarządza przejściem (ang. *transition*) VTL-a. Bezpieczne VTL-e zawsze działają w trybie długim (który jest modelem wykonania instrukcji przez procesory AMD64, gdzie procesor uzyskuje dostęp do instrukcji i rejestrów tylko 64-bitowych) z włączonym stronicowaniem. Każdy inny model wykonania nie jest obsługiwany. Upraszcza to uruchamianie i zarządzanie

bezpiecznymi VTL-ami, a także zapewnia dodatkowy poziom ochrony dla kodu działającego w trybie bezpiecznym. (Niektóre inne ważne konsekwencje są omówione w dalszej części rozdziału).

Dla uzyskania wydajności wirtualny procesor ma pewne rejestry, które są współdzielone między VTL-ami, i inne rejestry, które są prywatne dla każdego VTL-a. Stan współdzielonych rejestrów nie zmienia się podczas przełączania między VTL-ami. Pozwala to na szybkie przekazywanie niewielkiej ilości informacji między VTL-ami, a także zmniejsza narzut przełączania kontekstu podczas przełączania między VTL-ami. Każdy VTL ma swoją własną instancję prywatnych rejestrów, do których dostęp może mieć tylko ten VTL. Hipernadzorca obsługuje zapisywanie i przywracanie zawartości rejestrów prywatnych podczas przełączania między VTL-ami. Tak więc, gdy wchodzimy do VTL-a na procesorze wirtualnym, stan rejestrów prywatnych zawiera te same wartości co podczas ostatniego uruchomienia tego VTL-a przez procesor wirtualny.

Większość stanów rejestrów procesora wirtualnego jest współdzielona między VTL-ami. W szczególności rejestry ogólnego przeznaczenia, rejestry wektorowe i rejestry zmiennoprzecinkowe są współdzielone między wszystkimi VTL-ami z kilkoma wyjątkami, takimi jak rejestry RIP i RSP. Rejestry prywatne obejmują niektóre rejestry kontrolne, niektóre rejestry architektoniczne i wirtualne MSR-y hipernadzorcy. Mechanizm bezpiecznego przechwytywania (szczegóły w poprzednim punkcie) służy do umożliwienia środowisku bezpiecznemu kontrolowania, do których MSR-ów może mieć dostęp środowisko trybu normalnego. Tabela 9.3 podsumowuje, które rejestry są współdzielone między VTL-ami, a które są prywatne dla każdego VTL-a.

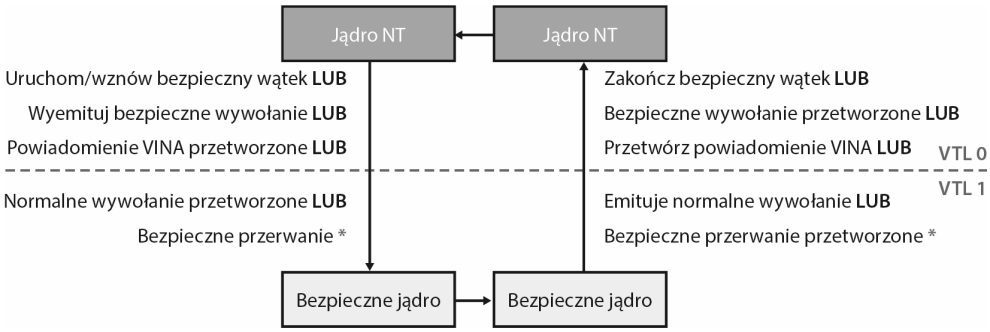
Bezpieczne wywołania

Gdy jądro NT potrzebuje usług świadczonych przez bezpieczne jądro, używa specjalnej funkcji *VslpEnterIumSecureMode*. Procedura przyjmuje 104-bajtową strukturę danych (zwaną *SKCALL*), która służy do opisania rodzaju operacji (wywołanie usługi, opróżnienie brokera czasu (TB — ang. *Time Broker*), wznowienie wątku lub wywołanie enklawy), numeru bezpiecznego wywołania oraz maksymalnie 12 parametrów 8-bajtowych. Funkcja podnosi poziom IRQL procesora, jeśli jest to konieczne, i określa wartość pliku *cookie* bezpiecznego wątku. Wartość ta informuje bezpieczne jądro, który bezpieczny wątek będzie przetwarzał żądanie. Następnie (ponownie) uruchamia pętlę wysyłania bezpiecznych wywołań. Stan wykonywalności każdego VTL-a jest maszyną stanów, która zależy od innych VTL-ów.

Pętla opisana przez funkcję *VslpEnterIumSecureMode* zarządza wszystkimi operacjami pokazanymi po lewej stronie rysunku 9.33 w VTL 0 (z wyjątkiem przypadku bezpiecznych przerwań). Jądro NT może podjąć decyzję o wejściu do bezpiecznego jądra, a bezpieczne jądro może zdecydować się na wejście do normalnego jądra NT. Pętla rozpoczyna się od wejścia do bezpiecznego jądra poprzez procedurę *HvlSwitchToVsmVtl1* (określającą operację żądaną przez wywołującego). Ta ostatnia funkcja, która jest zwracana tylko wtedy, gdy bezpieczne jądro zażąda przełączenia VTL-a, zapisuje wszystkie współdzielone rejestry i kopiuje całą strukturę danych *SKCALL* w niektórych dobrze zdefiniowanych rejestrach procesora, takich jak: rejestry RBX i SSE, od XMM10 do XMM15. Na koniec funkcja ta emituje hiperwywołanie *HvVtlCall* do hipernadzorcy. Hipernadzorca przełącza się na docelowy VTL (poprzez załadowanie struktury VMCS zapisanej dla każdego VTL-a) i zapisuje powód wejścia bezpiecznego wywołania VTL-a do strony kontrolnej VTL-a. W rzeczywistości, aby móc określić powód wejścia bezpiecznego VTL-a do bezpiecznego jądra, hipernadzorca utrzymuje informacyjną stronę pamięci, która jest współdzielona przez każdy bezpieczny VTL. Ta strona jest używana do dwukierunkowej komunikacji pomiędzy hipernadzorcą a kodem uruchomionym w bezpiecznym VTL-u na wirtualnym procesorze.

TABELA 9.3. *Stany rejestrów procesora wirtualnego ze względu na wirtualne poziomy zaufania VTL*

Typ	Rejestry ogólne	MSR-y
Współdzielony	Rax, Rbx, Rcx, Rdx, Rsi, Rdi, Rbp CR2 R8 – R15 DR0 – DR5 Stan punktu zmiennoprzecinkowego X87 Rejestry XMM Rejestry AVX Rejestr XCR0 (XFEM) DR6 (zależny od procesora)	HV_X64_MSR_TSC_FREQUENCY HV_X64_MSR_VP_INDEX HV_X64_MSR_VP_RUNTIME HV_X64_MSR_RESET HV_X64_MSR_TIME_REF_COUNT HV_X64_MSR_GUEST_IDLE HV_X64_MSR_DEBUG_DEVICE_OPTIONS HV_X64_MSR_BELOW_1MB_PAGE HV_X64_MSR_STATS_PARTITION_RETAIL_PAGE HV_X64_MSR_STATS_VP_RETAIL_PAGE Procedury MTRR i procedura PAT MCG_CAP MCG_STATUS
Prywatny	RIP, RSP RFLAGS CR0, CR3, CR4 DR7 IDTR, GDTR CS, DS, ES, FS, GS, SS, TR, LDTR TSC DR6 (zależny od procesora)	SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP, STAR, LSTAR, CSTAR, SFMASK, EFER, KERNEL_GSBASE, FS.BASE, GS.BASE HV_X64_MSR_HYPERCALL HV_X64_MSR_GUEST_OS_ID HV_X64_MSR_REFERENCE_TSC HV_X64_MSR_APIC_FREQUENCY HV_X64_MSR_EOI HV_X64_MSR_ICR HV_X64_MSR_TPR HV_X64_MSR_APIC_ASSIST_PAGE HV_X64_MSR_NPIEP_CONFIG HV_X64_MSR_SIRBP HV_X64_MSR_SCONTROL HV_X64_MSR_SVERSION HV_X64_MSR_SIEFP HV_X64_MSR_SIMP HV_X64_MSR_EOM HV_X64_MSR_SINTO – HV_X64_MSR_SINT15 HV_X64_MSR_STIMERO_CONFIG – HV_X64_MSR_STIMER3_CONFIG HV_X64_MSR_STIMERO_COUNT – HV_X64_MSR_STIMER3_COUNT Lokalne rejestry APIC (w tym CR8/TPR)



RYSUNEK 9.33. *Pętla rozsyłania w wirtualnym trybie bezpiecznym VSM*

Procesor wirtualny ponownie uruchamia wykonanie w kontekście VTL 1, w funkcji *SkCallNormalMode* bezpiecznego jądra. Kod odczytuje powód wejścia VTL-a; jeśli nie jest to bezpieczne przerwanie, ładuje bieżący blok sterowania procesorem SKPRCB (bezpieczne jądro), wybiera wątek, na którym ma być uruchomiony (zaczynając od pliku *cookie* bezpiecznego wątku), i kopiuje zawartość struktury danych *SKCALL* z rejestrów współdzielonych procesora do bufora pamięci. Na koniec wywołuje procedurę dyspozytora *IumInvokeSecureService*, która przetwarza żądane bezpieczne wywołanie, wysyłając je do właściwej funkcji (i implementuje część pętli rozsyłania w VTL 1).

Ważną koncepcją do zrozumienia jest to, że bezpieczne jądro może mapować i uzyskiwać dostęp do pamięci VTL 0, więc nie ma potrzeby marszalowania i kopiowania jakiejkolwiek docelowej struktury danych, wskazanej przez jeden lub więcej parametrów, do pamięci VTL 1. Ta koncepcja nie będzie miała zastosowania do zwykłego wywołania, co omówimy w następnej sekcji.

Jak widzieliśmy w poprzednim punkcie, bezpieczne przerwania (i przechwyty) są wysyłane przez hipernadzorcę, który uprzedza każdy kod wykonywany w VTL 0. W tym przypadku, gdy kod VTL 1 rozpoczyna wykonywanie, rozsyła przerwanie do właściwej procedury obsługi przerwania ISR. Gdy procedura ISR kończy swoje działanie, bezpieczne jądro natychmiast emituje hiperwywołanie *HvVtlReturn*. W rezultacie kod w VTL 0 wznawia wykonywanie w punkcie, w którym został wcześniej przerwany, a który nie znajduje się w pętli rozsyłania bezpiecznych wywołań. Dlatego bezpieczne przerwania nie są częścią pętli rozsyłania, nawet jeśli nadal produkują przełącznik VTL.

Wywołania normalne

Wywołania normalne są zarządzane podobnie jak wywołania bezpieczne (z analogiczną pętlą rozsyłania zlokalizowaną w VTL 1, zwaną *pętlą wywołań normalnych*), ale z kilkoma istotnymi różnicami:

- Wszystkie współdzielone rejestry VTL są bezpiecznie czyszczone przez bezpieczne jądro przed wyemitowaniem *HvVtlReturn* do hipernadzorcy w celu przełączenia VTL-a. Zapobiega to wyciekowi jakichkolwiek bezpiecznych danych do trybu normalnego.
- Normalne jądro NT nie może czytać bezpiecznej pamięci VTL 1. Do prawidłowego przekazania parametrów *syscall* i struktur danych potrzebnych do normalnego wywołania wymagany jest bufor pamięci, który mogą współdzielić zarówno bezpieczne jądro, jak i normalne jądro. Bezpieczne jądro przydziela ten wspólny bufor za pomocą normalnego wywołania `ALLOCATE_VM` (które nie wymaga przekazania żadnego wskaźnika jako parametru).

To ostatnie jest wysyłane do funkcji *MmAllocateVirtualMemory* w normalnym jądrze NT. Przydzielona pamięć jest w bezpiecznym jądrze ponownie mapowana pod tym samym adresem wirtualnym i staje się częścią wspólnej puli pamięci procesu bezpiecznego.

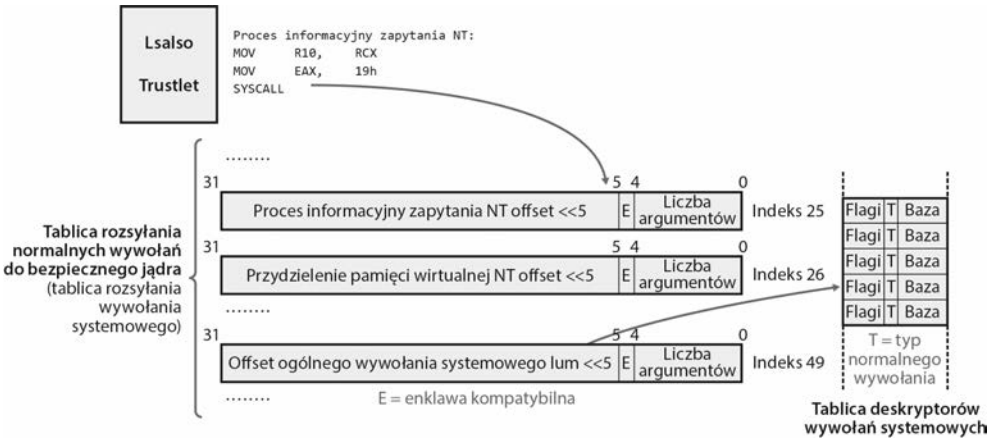
- Jak omówimy w dalszej części rozdziału, tryb IUM został pierwotnie zaprojektowany tak, aby umożliwić wykonywanie specjalnych programów wykonywalnych Win32, które powinny były moc działać zarówno w normalnym świecie, jak i w świecie bezpiecznym. Standardowe niezmodyfikowane biblioteki *Ntdll.dll* i *KernelBase.dll* są mapowane nawet w trybie izolowanego użytkownika IUM. Fakt ten ma istotne następstwa w postaci konieczności zastępowania przez bezpieczne jądro prawie wszystkich natywnych API jądra NT (od których zależy *Kernel32.dll* i wiele innych bibliotek trybu użytkownika).

Aby poprawnie poradzić sobie z opisanymi problemami, bezpieczne jądro zawiera marszaler, który identyfikuje i poprawnie kopiuje struktury danych, wskazywane przez parametry API jądra NT, do współdzielonego bufora. Marszaler jest również w stanie określić rozmiar współdzielonego bufora, który zostanie przydzielony z bezpiecznej puli pamięci procesu. Bezpieczne jądro definiuje trzy rodzaje zwykłych wywołań:

- **Wyłączone normalne wywołanie** nie jest zaimplementowane w bezpiecznym jądrze i jeśli zostanie wywołane z IUM, po prostu kończy się niepowodzeniem z kodem wyjścia `STATUS_INVALID_SYSTEM_SERVICE`. Tego rodzaju wywołanie nie może być wywołane bezpośrednio przez samo bezpieczne jądro.
- **Włączone normalne wywołanie** jest zaimplementowane tylko w jądrze NT i jest możliwe do wywołania z trybu izolowanego użytkownika IUM w jego oryginalnej wersji *Nt* lub *Zw* (poprzez *Ntdll.dll*). Nawet bezpieczne jądro może zażądać włączonego normalnego wywołania, ale tylko poprzez mały kod stub, który ładuje numer normalnego wywołania, ustawia najwyższy bit w numerze i wywołuje procedurę rozsyłania normalnych wywołań (*IumGenericSyscall*). Najwyższy bit identyfikuje normalne wywołanie jako wymagane przez samo bezpieczne jądro, a nie przez moduł *Ntdll.dll* załadowany w trybie izolowanego użytkownika IUM.
- **Specjalne normalne wywołanie** jest zaimplementowane częściowo lub całkowicie w bezpiecznym jądrze (VTL 1), które może filtrować wyniki oryginalnej funkcji lub całkowicie przeprojektować jej kod.

Włączone i specjalne normalne wywołania mogą być oznaczone jako *KernelOnly* (wyłącznie jądro). W tym drugim przypadku normalne wywołanie może być żądane tylko od samego bezpiecznego jądra (a nie od bezpiecznych procesów). Listę włączonych i specjalnych normalnych wywołań (które można wywołać z oprogramowania uruchomionego w wirtualnym trybie bezpiecznym VSM) przedstawiliśmy już w rozdziale 3. części I, w punkcie „Wywołania systemowe dostępne dla programów Trustlet".

Rysunek 9.34 przedstawia przykład specjalnego normalnego wywołania. W tym przykładzie trustlet LsaIso wywołał natywne API *NtQueryInformationProcess* w celu zażądania informacji o określonym procesie. Plik *Ntdll.dll* zmapowany w trybie izolowanego użytkownika IUM przygotowuje numer wywołania systemowego syscall i wykonuje instrukcję `SYSCALL`, która przekazuje przepływ wykonania do globalnego dyspozytora wywołań systemowych *KiSystemServiceStart*, rezydującego w bezpiecznym jądrze (VTL 1). Globalny dyspozytor wywołań systemowych rozpoznaje, że numer wywołania systemowego należy do normalnego wywołania i używa numeru, aby uzyskać dostęp do tablicy `IumSyscallDispatchTable`, która odzwierciedla tablicę rozsyłania normalnych wywołań.

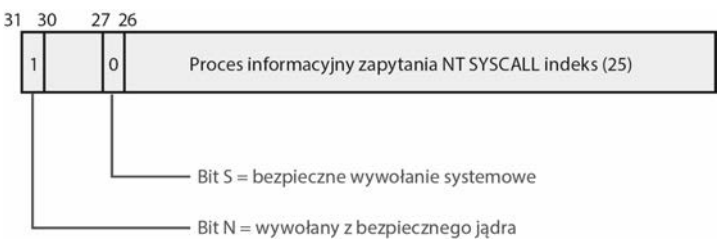


RYSUNEK 9.34. *Trustlet wykonujący specjalne normalne wywołanie do API procesu NtQueryInformationProcess*

Tablica rozsyłania normalnych wywołań zawiera tablicę zagęszczonych wpisów, które są generowane w fazie 0 uruchamiania bezpiecznego jądra (co omówiono w dalszej części tego rozdziału). Każdy wpis zawiera offset do funkcji docelowej (obliczony względem samej tablicy) i liczbę jej argumentów (z pewnymi flagami). Wszystkie offsety w tablicy są początkowo obliczone tak, aby wskazywały na procedurę rozsyłania normalnych wywołań (*IumGenericSyscall*). Po pierwszym cyklu inicjalizacji procedura startowa bezpiecznego jądra łata każdy wpis, który odzwierciedla specjalne wywołanie. Nowy offset jest wskazywany na część kodu, która implementuje normalne wywołanie w bezpiecznym jądrze.

W rezultacie tego na rysunku 9.34 globalny dyspozytor wywołań systemowych przekazuje wykonanie do części funkcji *NtQueryInformationProcess* zaimplementowanej w bezpiecznym jądrze. Ta ostatnia sprawdza, czy żądana klasa informacji jest jednym z małych podzbiorów prezentowanych w bezpiecznym jądrze, a jeśli tak, to czy używa małego kodu stub do wywołania procedury rozsyłania normalnych wywołań (*IumGenericSyscall*).

Rysunek 9.35 pokazuje numer selektora syscall dla API *NtQueryInformationProcess*. Zauważ, że stub ustawia najwyższy bit (bit *N*) numeru syscall, aby wskazać, że normalne wywołanie jest żądane przez bezpieczne jądro. Dyspozytor normalnych wywołań sprawdza parametry i wywołuje marszalera, który jest w stanie zamarszalować każdy argument i skopiować go we właściwym offsecie współdzielonego bufora. W selektorze znajduje się jeszcze jeden bit, który dodatkowo rozróżnia normalne wywołanie lub bezpieczne wywołanie systemowe, co zostało omówione w dalszej części tego rozdziału.



RYSUNEK 9.35. *Numer selektora Syscall w bezpiecznym jądrze*

Marszaler działa dzięki dwóm ważnym tablicom, które opisują każde normalne wywołanie: tablicy deskryptorów (pokazanej po prawej stronie rysunku 9.34) i tablicy deskryptorów argumentów. Z tych tablic marszaler może pobrać wszystkie potrzebne mu informacje: typ normalnego wywołania, indeks funkcji marszalingu, typ argumentu, rozmiar oraz typ danych, na które wskazuje (jeśli argument jest wskaźnikiem).

Po prawidłowym wypełnieniu bufora współdzielonego przez marszalera bezpieczne jądro kompiluje strukturę danych *SKCALL* i wchodzi w pętlę rozsyłania normalnych wywołań (*SkCallNormalMode*). Ta część pętli zapisuje i czyści wszystkie współdzielone rejestry wirtualne CPU, wyłącza przerwania i przenosi kontekst wątku do wątku PRCB (więcej o planowaniu wątków w dalszej części rozdziału). Następnie kopiuje zawartość struktury danych *SKCALL* do jakiegoś współdzielonego rejestru. W ostatnim etapie wywołuje hipernadzorcę za pomocą hiperwywołania *HvVtlReturn*.

Następnie wznawia wykonywanie kodu w pętli bezpiecznego wysyłania połączeń w VTL 0. Jeśli w kolejce znajdują się jakieś oczekujące przerwania, są one przetwarzane w normalny sposób (tylko jeśli IRQL na to pozwala). Pętla rozpoznaje żądanie normalnej obsługi połączenia i wywołuje funkcję *NtQueryInformationProcess* zaimplementowaną w VTL 0. Po zakończeniu przetwarzania przez tę ostatnią funkcję pętla uruchamia się ponownie i ponownie wchodzi do bezpiecznego jądra (jak w przypadku bezpiecznych wywołań), nadal poprzez procedurę *HvlSwitchToVsmVtl1*, ale z żądaniem wykonania innej operacji: *Przywróć wątek*. To, jak sama nazwa wskazuje, pozwala bezpiecznemu jądru przełączyć się na pierwotny bezpieczny wątek i kontynuować wykonanie instrukcji, która została wcześniej zawieszona w celu wykonania normalnego wywołania.

Implementacja włączonych normalnych wywołań jest taka sama z wyjątkiem tego, że wywołania te mają swoje wpisy w tablicy rozsyłania normalnych wywołań, które wskazują bezpośrednio na procedurę rozsyłania normalnych wywołań, *IumGenericSyscall*. W ten sposób kod zostanie przekazany bezpośrednio do handlera, pomijając jakikolwiek kod implementujący API w bezpiecznym jądrze.

Bezpieczne wywołania systemowe

Ostatni typ wywołań systemowych dostępnych w bezpiecznym jądrze jest podobny do standardowych wywołań systemowych dostarczanych przez jądro NT do oprogramowania trybu użytkownika VTL 0. Bezpieczne wywołania systemowe są wykorzystywane do świadczenia usług tylko dla bezpiecznych procesów (trustletów). Oprogramowanie VTL 0 nie może w żaden sposób emitować bezpiecznych wywołań systemowych. Jak omówimy to w punkcie „Tryb izolowanego użytkownika" w dalszej części tego rozdziału, każdy trustlet mapuje DLL warstwy natywnej IUM (*Iumdll.dll*) w swojej przestrzeni adresowej. Plik *Iumdll.dll* ma takie samo zadanie jak jego odpowiednik w VTL 0, plik *Ntdll.dll* — implementuje natywne funkcje stub wywołań systemowych (syscall) dla aplikacji trybu użytkownika. Stub kopiuje numer wywołania systemowego syscall w rejestrze i emituje instrukcję `SYSCALL` (instrukcja używa różnych kodów operacyjnych (opcodes) w zależności od platformy).

Numery bezpiecznych wywołań systemowych zawsze mają 28. bit ustawiony na 1 (w architekturach AMD64, natomiast ARM64 używa 16. bitu). W ten sposób globalny dyspozytor wywołań systemowych (*KiSystemServiceStart*) rozpoznaje, że numer wywołania syscall należy do bezpiecznego wywołania systemowego (a nie do zwykłego wywołania) i przełącza się do tablicy *SkiSecureServiceTable*, która odzwierciedla tablicę rozsyłania bezpiecznych wywołań systemowych. Jak w przypadku zwykłych wywołań, globalny dyspozytor sprawdza, czy numer wywołania mieści się w limicie, przydziela miejsce na stosie dla argumentów (jeśli jest to potrzebne), oblicza adres końcowy wywołania systemowego i przekazuje do niego wykonanie kodu.

W sumie wykonanie kodu pozostaje w VTL 1, ale aktualny poziom uprawnień procesora wirtualnego wzrasta z 3 (tryb użytkownika) do 0 (tryb jądra). Tablica rozsyłania bezpiecznych wywołań systemowych jest zagęszczana — podobnie jak tablica rozsyłania normalnych wywołań — w fazie 0 uruchamiania bezpiecznego jądra. Jednakże wpisy w tej tablicy są wszystkie ważne i wskazują na funkcje zaimplementowane w bezpiecznym jądrze.

Bezpieczne wątki i planowanie

Jak opiszemy w punkcie „Tryb izolowanego użytkownika", jednostkami wykonawczymi w VSM są bezpieczne wątki, które żyją w przestrzeni adresowej opisanej przez bezpieczny proces. Bezpieczne wątki mogą być wątkami trybu jądra lub trybu użytkownika. VSM utrzymuje ścisłą zgodność pomiędzy każdym bezpiecznym wątkiem trybu użytkownika a normalnym wątkiem żyjącym w VTL 0.

W istocie planowanie wątków bezpiecznego jądra zależy całkowicie od normalnego jądra NT; bezpieczne jądro nie zawiera zastrzeżonego planisty (z założenia powierzchnia bezpiecznego jądra narażona na atak musi być niewielka). W rozdziale 3. części I opisaliśmy, jak jądro NT tworzy proces i odpowiedni wątek początkowy. W punkcie opisującym etap 4, „Tworzenie początkowego wątku oraz jego stosu i kontekstu", wyjaśniamy, że tworzenie wątku odbywa się w dwóch częściach:

- Tworzony jest obiekt wątku wykonawczego; przydzielany jest jego stos jądra i użytkownika. Procedura *KeInitThread* jest wywoływana w celu ustawienia początkowego kontekstu wątku dla wątków trybu użytkownika. *KiStartUserThread* jest pierwszą procedurą, która zostanie wykonana w kontekście nowego wątku, która obniży IRQL wątku i wywoła *PspUserThreadStartup*.
- Kontrola wykonania jest następnie zwracana do procesu *NtCreateUserProcess*, który w dalszym etapie wywołuje wątek *PspInsertThread* w celu zakończenia inicjalizacji wątku i wstawienia go do przestrzeni nazw menedżera obiektów.

W ramach swojej pracy, gdy *PspInsertThread* wykryje, że wątek należy do bezpiecznego procesu, wywołuje *VslCreateSecureThread*, który — jak sama nazwa wskazuje — wykorzystuje wywołanie usługi bezpiecznej *Utwórz wątek* (ang. *Create Thread*), aby zwrócić się do bezpiecznego jądra o utworzenie powiązanego bezpiecznego wątku. Bezpieczne jądro weryfikuje parametry i pobiera strukturę danych bezpiecznego obrazu procesu (więcej szczegółów w dalszej części rozdziału). Następnie przydziela obiekt bezpiecznego wątku i jego blok środowiskowy TEB, tworzy początkowy kontekst wątku (pierwszą procedurą, która zostanie uruchomiona jest *SkpUserThreadStartup*), a na koniec czyni wątek możliwym do zaplanowania. Ponadto handler usługi bezpiecznej w VTL 1 po oznaczeniu wątku jako gotowego do uruchomienia zwraca określony plik *cookie* wątku, który jest przechowywany w strukturze danych `ETHREAD`.

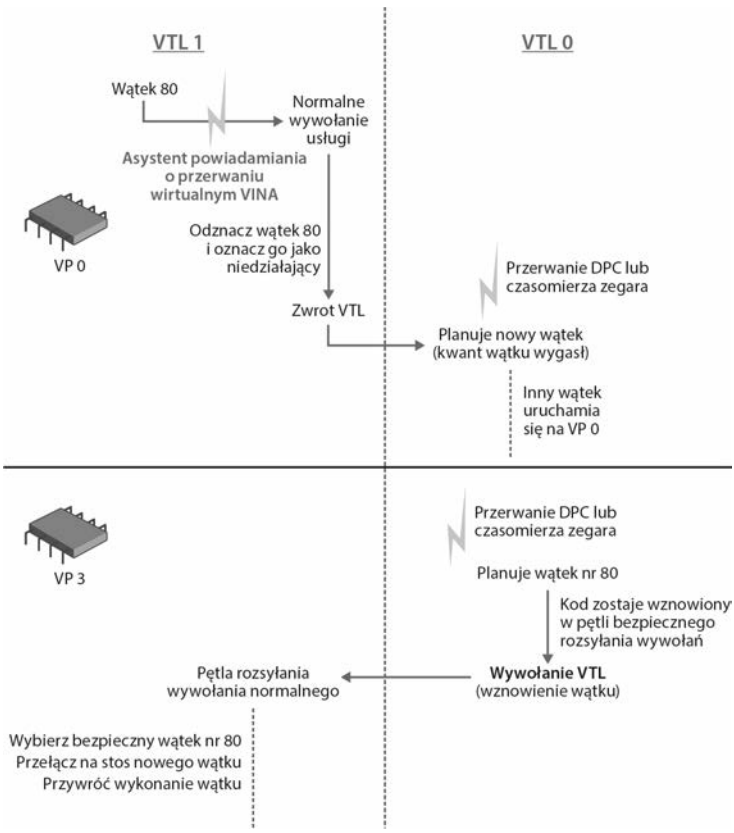
Nowy bezpieczny wątek nadal rozpoczyna się w VTL 0. Jak opisano w punkcie „Etap 7" w rozdziale 3. części I, *PspUserThreadStartup* wykonuje ostateczną inicjalizację wątku użytkownika w nowym kontekście. W przypadku, gdy stwierdzi, że procesem właścicielskim wątku jest trustlet, *PspUserThreadStartup* wywołuje funkcję *VslStartSecureThread*, która odwołuje się do pętli rozsyłania bezpiecznych wywołań poprzez procedurę *VslpEnterIumSecureMode* w VTL 0 (przekazując plik *cookie* bezpiecznego wątku zwrócony przez handlera bezpiecznej usługi *Utwórz wątek* (ang. *Create Thread*)). Pierwszą operacją, o którą pętla dyspozycyjna zwraca się do bezpiecznego jądra, jest wznowienie wykonywania bezpiecznego wątku (nadal poprzez hiperwywołanie *HvVtlCall*).

Przed przełączeniem na VTL 0 bezpieczne jądro wykonywało kod w pętli rozsyłania normalnych wywołań (*SkCallNormalMode*). Hiperwywołanie wykonane przez normalne jądro uruchamia ponownie wykonanie (hiperwywołania) w tej samej procedurze pętli. Pętla rozsyłania

VTL 1 rozpoznaje żądanie wznowienia nowego wątku; przełącza swój kontekst wykonawczy na nowy bezpieczny wątek, dołącza się do jego przestrzeni adresowej i umożliwia jego uruchomienie. W ramach przełączania kontekstu wybierany jest nowy stos (który został wcześniej zainicjalizowany przez bezpieczne wywołanie *Utwórz wątek* (ang. *Create Thread*)). Zawiera on adres pierwszej bezpiecznej funkcji systemowej wątku, *SkpUserThreadStartup*, która — podobnie jak w przypadku zwykłych wątków NT — ustawia początkowy kontekst „thunk", aby uruchomić procedurę inicjalizacji programu ładującego obraz (*LdrInitializeThunk* w pliku *Ntdll.dll*).

Po uruchomieniu nowy bezpieczny wątek może powrócić do normalnego trybu z dwóch głównych powodów: emituje normalne wywołanie, które musi być przetworzone w VTL 0, lub przerwania VINA uprzedzają wykonanie kodu. Mimo że te dwa przypadki są przetwarzane w nieco inny sposób, oba skutkują wykonaniem pętli rozsyłania normalnych wywołań (*SkCallNormalMode*).

Jak już wcześniej omówiono w części I, w rozdziale 4., „Wątki", planista NT działa dzięki zegarowi procesora, który generuje przerwanie za każdym razem, gdy zegar systemowy „wystrzeli" (zwykle co 15,6 milisekundy). Procedura obsługi przerwania zegara uaktualnia czasy procesora i oblicza, czy upływa kwant wątku. Przerwanie jest skierowane do VTL 0, więc gdy wirtualny procesor wykonuje kod w VTL 1, hipernadzorca wstrzykuje do bezpiecznego jądra przerwanie VINA, co pokazano na rysunku 9.36. Przerwanie VINA wyłącza aktualnie wykonywany kod, obniża IRQL do wartości IRQL poprzedniego, wyłączonego kodu i emituje normalne wywołanie VINA dla wejścia do VTL 0.



RYSUNEK 9.36. *Schemat planowania wątków bezpiecznych*

W ramach standardowego procesu rozsyłania normalnych wywołań, zanim bezpieczne jądro wyemituje hiperwywołanie *HvVtlReturn, odłącza ono bieżący wątek wykonawczy od bloku sterującego wirtualnego procesora PRCB*. Jest to ważne: procesor wirtualny w VTL 1 nie jest już związany z żadnym kontekstem wątku i w następnym cyklu pętli bezpieczne jądro może przełączyć się na inny wątek lub zdecydować o zmianie harmonogramu wykonania bieżącego wątku.

Po przełączeniu VTL-a jądro NT wznawia wykonywanie w pętli rozsyłania normalnych wywołań i kontynuuje to wykonywanie w kontekście nowego wątku. Zanim jednak ma szansę wykonać jakikolwiek kod, zostaje on uprzedzony przez procedurę obsługi przerwania zegara, która może obliczyć nową wartość kwantu i jeśli ta wygasła, przełączyć wykonanie innego wątku. Gdy nastąpi przełączenie kontekstu i inny wątek wejdzie do VTL 1, pętla rozsyłania normalnych wywołań planuje inny bezpieczny wątek w zależności od wartości pliku *cookie* bezpiecznego wątku:

- Bezpieczny wątek z puli bezpiecznych wątków, jeśli normalne jądro NT weszło do VTL 1 w celu wysłania bezpiecznego połączenia (w tym przypadku plik *cookie* bezpiecznego wątku ma wartość 0).
- Nowo utworzony bezpieczny wątek, jeżeli wątek został przesunięty przez planistę do wykonania (plik *cookie* bezpiecznego wątku ma wartość prawidłową). Jak widać na rysunku 9.36, nowy wątek może być również przeorganizowany przez inny procesor wirtualny (na przykładzie procesora wirtualnego VP 3).

W opisanym schemacie wszystkie decyzje dotyczące planowania są wykonywane tylko w VTL 0. Pętla bezpiecznych wywołań i pętla normalnych wywołań współpracują ze sobą, aby w VTL 1 prawidłowo przełączyć kontekst bezpiecznego wątku. Wszystkie bezpieczne wątki mają przypisany wątek w normalnym jądrze. Nie jest jednak odwrotnie; jeśli normalny wątek w VTL 0 zdecyduje się na emisję bezpiecznego wywołania, bezpieczne jądro wysyła żądanie za pomocą dowolnego kontekstu wątku z puli wątków.

Integralność kodu wymuszona przez hipernadzorcę

Integralność kodu wymuszona przez hipernadzorcę (HVCI — ang. *Hypervisor Enforced Code Integrity*) jest funkcją, która zasila strażnika urządzeń (ang. *Device Guard*) i zapewnia charakterystykę *W^X* (wymawiane *W xor X*) pamięci jądra VTL 0. Jądro NT nie może mapować i wykonywać operacji na dowolnym typie pamięci wykonywalnej w trybie jądra, bez pomocy bezpiecznego jądra. Bezpieczne jądro pozwala na uruchamianie w jądrze maszyny tylko właściwych, cyfrowo podpisanych sterowników. Jak omówimy to w następnym rozdziale, bezpieczne jądro śledzi każdą stronę wirtualną przydzieloną w normalnym jądrze NT; strony pamięci oznaczone w jądrze NT jako wykonywalne są uważane za *strony uprzywilejowane*. Tylko bezpieczne jądro może w nich dokonywać wpisów po poprawnym zweryfikowaniu ich zawartości przez moduł bezpiecznej integralności kodu jądra (SKCI — ang. *Secure Kernel Code Integrity*).

Więcej o HVCI możesz przeczytać w rozdziale 7. części I, w punktach „Ochrona urządzenia” i „Ochrona poświadczeń”.

Wirtualizacja uruchomieniowa UEFI

Kolejną usługą świadczoną przez bezpieczne jądro (gdy integralność HVCI jest włączona) jest możliwość wirtualizacji i ochrony usług uruchomieniowych UEFI. Jak omawiamy to w rozdziale 12., usługi oprogramowania układowego firmware UEFI są głównie implementowane przy użyciu dużej tablicy wskaźników funkcji. Część tablicy zostanie usunięta z pamięci po przejęciu kontroli

przez system operacyjny i wywołaniu funkcji `ExitBootServices`, ale inna część tablicy, odzwierciedlająca usługi uruchomieniowe (*Runtime*), pozostanie zmapowana nawet po tym, jak system operacyjny przejmie już pełną kontrolę nad maszyną. W rzeczywistości jest to konieczne, ponieważ czasami system operacyjny musi wchodzić w interakcje z konfiguracją i usługami UEFI.

Każdy producent sprzętu implementuje własne oprogramowanie układowe firmware UEFI. W przypadku HVCI firmware powinien współpracować w celu zapewnienia niezapisywalnego stanu każdej ze swoich stron pamięci wykonywalnej (żadna strona oprogramowania układowego firmware nie może być mapowana w VTL 0 ze stanem odczyt, zapis i wykonanie). Zakres pamięci, w którym rezyduje oprogramowanie układowe firmware UEFI, jest opisany przez wiele struktur danych *MEMORY_DESCRIPTOR*, umieszczonych w mapie pamięci EFI. Windows Loader parsuje te dane w celu właściwego zabezpieczenia pamięci firmware'u UEFI. Niestety w oryginalnej implementacji UEFI kod i dane były przechowywane w jednej sekcji (lub wielu sekcjach) i były opisane przez odpowiednie deskryptory pamięci. Ponadto niektóre sterowniki urządzeń odczytywały lub zapisywały dane konfiguracyjne bezpośrednio z regionów pamięci UEFI. To w oczywisty sposób nie było zgodne z HVCI.

Aby przezwyciężyć ten problem, bezpieczne jądro stosuje następujące dwie strategie:

- Nowe wersje oprogramowania układowego firmware UEFI (zgodne ze specyfikacją UEFI 2.6 i wyższą) utrzymują nową tablicę konfiguracyjną (zlinkowaną z tablicą usług rozruchowych), zwaną tablicą atrybutów pamięci (MAT — ang. *Memory Attribute Table*). Tablica MAT definiuje drobnoziarniste sekcje regionu pamięci UEFI, które są podsekcjami deskryptorów pamięci zdefiniowanych przez mapę pamięci EFI. Każda sekcja nigdy nie posiada równocześnie atrybutu ochrony przed wykonaniem, jak i przed zapisem.
- W przypadku starego oprogramowania układowego firmware'u bezpieczne jądro mapuje w VTL 0 całą pamięć fizyczną regionu UEFI firmware z prawem dostępu tylko do odczytu.

W ramach pierwszej strategii podczas rozruchu systemu Windows Loader łączy informacje znajdujące się zarówno w mapie pamięci EFI, jak i w MAT, tworząc tablicę deskryptorów pamięci, które dokładnie opisują cały region firmware'u. Następnie kopiuje je do zarezerwowanego bufora znajdującego się w VTL 1 (używanego w ścieżce hibernacji) i sprawdza, czy każda sekcja firmware'u nie narusza założenia *W^X*. Jeśli tak, to podczas startu bezpieczne jądro stosuje odpowiednią ochronę SLAT dla każdej strony, która należy do bazowego regionu firmware'u UEFI. Strony fizyczne są chronione przez SLAT, ale ich wirtualna przestrzeń adresowa w VTL 0 jest nadal w całości oznaczona jako RWX. Zachowanie ochrony RWX pamięci wirtualnej jest ważne, ponieważ bezpieczne jądro musi obsługiwać wznawianie z hibernacji w scenariuszu, w którym ochrona zastosowana we wpisach MAT może się zmienić. Ponadto zachowuje to kompatybilność ze starszymi sterownikami, które odczytują lub zapisują bezpośrednio z regionu pamięci UEFI, zakładając, że zapis jest wykonywany w odpowiednich sekcjach. (Również kod UEFI powinien być w stanie dokonywać wpisów w swojej własnej pamięci, która jest mapowana w VTL 0). Ta strategia pozwala bezpiecznemu jądru uniknąć mapowania jakiegokolwiek kodu firmware'u w VTL 1; jedyną częścią firmware'u, która pozostaje w VTL 1 jest sama tablica funkcji uruchomieniowych. Zachowanie tablicy w VTL 1 pozwala kodowi przywracającemu maszynę ze stanu hibernacji bezpośrednio aktualizować wskaźnik funkcji usług uruchomieniowych UEFI.

Ta druga strategia nie jest optymalna i jest używana tylko do umożliwienia starym systemom działania z włączonym HVCI. Kiedy bezpieczne jądro nie znajdzie żadnej tablicy MAT w oprogramowaniu układowym firmware, nie ma innego wyjścia jak tylko zmapować cały kod usług uruchomieniowych UEFI w VTL 1. Historycznie wiele błędów zostało odkrytych w kodzie UEFI oprogramowania układowego firmware (w szczególności w SMM).

Mapowanie oprogramowania układowego firmware w VTL 1 może być niebezpieczne, ale jest to jedyne rozwiązanie zgodne z HVCI. (Nowe systemy, jak podano wcześniej, nigdy nie mapują żadnego kodu UEFI oprogramowania układowego firmware w VTL 1). W czasie startu systemu HAL jądra NT wykrywa, że HVCI jest włączone i że oprogramowanie układowe firmware jest w całości zmapowane w VTL 1. Zmienia więc wskaźnik swojej wewnętrznej tablicy usług EFI na nową tablicę, zwaną tablicą wrappera UEFI. Wpisy tablicy wrappera zawierają procedury stub, które używają bezpiecznego wywołania `INVOKE_EFI_RUNTIME_SERVICE`, aby wejść do VTL 1. Bezpieczne jądro marszaluje parametry, wykonuje wywołanie oprogramowania układowego firmware i przekazuje wyniki do VTL 0. W tym przypadku cała pamięć fizyczna, która opisuje całe oprogramowanie układowe firmware UEFI jest nadal mapowana w trybie tylko do odczytu w VTL 0. Celem jest umożliwienie sterownikom prawidłowego odczytu informacji z regionu pamięci oprogramowania układowego firmware UEFI (jak na przykład tablice ACPI). Stare sterowniki, które dokonują wpisów bezpośrednio w regionach pamięci UEFI, nie są w tym scenariuszu zgodne z HVCI.

Kiedy bezpieczne jądro wznawia pracę z hibernacji, aktualizuje tablicę usług UEFI w pamięci, aby wskazać lokalizację nowych usług. Ponadto w systemach, które posiadają nowe oprogramowanie układowe firmware UEFI, bezpieczne jądro ponownie stosuje ochronę SLAT na każdym regionie pamięci zmapowanym w VTL 0 (Windows Loader jest w stanie zmienić wirtualne adresy regionów, jeśli jest to konieczne).

Uruchamianie VSM

Chociaż w rozdziale 12. opisujemy cały mechanizm uruchamiania i zamykania systemu Windows, w tym rozdziale opisano sposób uruchamiania bezpiecznego jądra i całej infrastruktury VSM. Bezpieczne jądro jest zależne od hipernadzorcy, programu Windows Loader i jądra NT, aby móc się prawidłowo uruchomić. W rozdziale 12. omawiamy Windows Loader, loader hipernadzorcy i wstępne fazy, w których bezpieczne jądro jest inicjalizowane w VTL 0 przez te dwa moduły. W tej sekcji skupimy się na metodzie uruchamiania VSM, która jest zaimplementowana w binarnym pliku *securekernel.exe*.

Pierwszy kod wykonywany przez plik binarny securekernel.exe wciąż działa w VTL 0; hipernadzorca został już uruchomiony, a tablice stron używane w VTL 1 zostały zbudowane. Bezpieczne jądro inicjalizuje następujące komponenty w VTL 0:

- Funkcja inicjalizacji menedżera pamięci przechowuje PFN struktury VTL 0 na poziomie głównym (*root*) i poziomie stron, zapisuje dane o integralności kodu oraz umożliwia HVCI, Kontrolę wykonania w oparciu o tryb MBEC (ang. *Mode-Based Execution Control*), CFG jądra i łatanie „na gorąco”.
- Współdzielone komponenty CPU specyficzne dla architektury, takie jak GDT i IDT.
- Wywołania normalne oraz tablice rozsyłania bezpiecznych wywołań systemowych (inicjalizacja i kompaktowanie).
- Procesor rozruchowy (*boot*) Proces uruchamiania procesora startowego wymaga od bezpiecznego jądra przydzielenia stosu jądra i przerwań, inicjalizowania specyficznych dla architektury komponentów, które nie mogą być współdzielone między różnymi procesorami (jak TSS) i wreszcie przydzielenia SKPRCB procesora. To ostatnie jest ważną strukturą danych, która, podobnie jak struktura danych PRCB w VTL 0, służy do przechowywania ważnych informacji związanych z każdym procesorem.

Kod inicjalizacyjny bezpiecznego jądra jest gotowy do wejścia po raz pierwszy do VTL 1. Funkcja inicjalizacji podsystemu hipernadzorcy (procedura *ShvlInitSystem*) łączy się z hipernadzorcą (poprzez klasy CPUID hipernadzorcy; więcej szczegółów w poprzednim rozdziale) i sprawdza obsługiwane rozszerzenia „oświecające". Następnie zapisuje tablicę stron VTL 1 (utworzoną wcześniej przez Windows Loader) i przydzielone strony hiperwywołań (używane do przechowywania parametrów hiperwywołań). Na koniec inicjalizuje i wprowadza VTL 1 w następujący sposób:

1. Włącza VTL 1 dla bieżącej partycji hipernadzorcy poprzez hiperwywołanie *HvEnablePartitionVtl*. Hipernadzorca kopiuje istniejącą tablicę SLAT normalnego VTL do VTL 1 i włącza MBEC oraz nowy VTL 1 dla partycji.
2. Włącza VTL 1 dla procesora startowego poprzez hiperwywołanie *HvEnableVpVtl*. Hipernadzorca inicjalizuje nową strukturę danych VMCS dla każdego poziomu, kompiluje ją i ustawia tablicę SLAT.
3. Następnie pyta hipernadzorcę o lokalizację zależnego od platformy kodu hiperwywołania *VtlCall* i *VtlReturn*. Kody operacyjne CPU potrzebne do wykonania wywołań VSM są ukryte przed implementacją bezpiecznego jądra. Dzięki temu większość kodu bezpiecznego jądra jest niezależna od platformy. Na koniec bezpieczne jądro wykonuje przejście do VTL 1, poprzez hiperwywołanie *HvVtlCall*. Hipernadzorca ładuje VMCS dla nowego VTL-a i przełącza się na niego (czyniąc go aktywnym). To w zasadzie sprawia, że nowy VTL może być uruchomiony.

Bezpieczne jądro rozpoczyna w VTL 1 złożoną procedurę inicjalizacji, która nadal zależy od programu Windows Loader, a także od jądra NT. Warto zauważyć, że na tym etapie pamięć VTL 1 jest nadal mapowana tożsamościowo w VTL 0; bezpieczne jądro i zależne od niego moduły są nadal dostępne dla normalnego świata. Po przełączeniu się do VTL 1, bezpieczne jądro inicjalizuje procesor startowy:

1. Uzyskuje wirtualny adres strony współdzielonej Syntetycznego kontrolera przerwań, licznika znaczników czasu TSC i strony pomocniczej procesora wirtualnego, które są dostarczane przez hipernadzorcę do współdzielenia danych między hipernadzorcą a kodem VTL 1. Mapuje w VTL 1 Stronę hiperwywołań.
2. Blokuje możliwość uruchomienia innych systemowych procesorów wirtualnych przez niższy VTL i żąda zerowego wypełnienia pamięci przy ponownym uruchomieniu do hipernadzorcy.
3. Inicjalizuje i wypełnia tablicę deskryptorów przerwań (IDT — ang. *Interrupt Descriptor Table*) procesora startowego. Konfiguruje IPI, wywołania zwrotne i obsługę przerwań bezpiecznego czasomierza oraz ustawia bieżący bezpieczny wątek jako domyślny wątek SKPRCB.
4. Uruchamia menedżera bezpiecznej pamięci VTL 1, który tworzy mapowanie tablicy startowej i mapuje pamięć programu ładującego w VTL 1, tworzy bezpieczną bazę PFN i hiperprzestrzeń systemową, inicjalizuje obsługę bezpiecznej puli pamięci i odczytuje blok ładujący VTL 0 w celu skopiowania deskryptorów modułów zaimportowanych obrazów bezpiecznego jądra (*Skci.dll*, *Cnf.sys* i *Vmsvcext.sys*). Na koniec, w celu ustalenia stanu każdego ze sterowników, przegląda listę załadowanych modułów NT, tworząc dla każdego z nich strukturę danych o normalnym zakresie adresów (NAR — ang. *normal address range*) oraz kompilując normalny wpis w tablicy (NTE — ang. *Normal Table Entry*) dla każdej strony składającej się na sekcje sterownika startowego. Ponadto funkcja inicjalizacji bezpiecznego menedżera pamięci stosuje prawidłową ochronę SLAT VTL 0 do każdej sekcji sterownika.

5. Inicjalizuje HAL, pulę bezpiecznych wątków, podsystem procesów, syntetyczny APIC, bezpieczne PNP oraz bezpieczne PCI.
6. Stosuje ochronę VTL 0 SLAT typu tylko do odczytu dla stron bezpiecznego jądra, konfiguruje MBEC i włącza wirtualne przerwanie VINA na procesorze startowym.

Kiedy ta część inicjalizacji się kończy, bezpieczne jądro usuwa mapowanie pamięci załadowanej przez procesor startowy. Bezpieczny menedżer pamięci, jak omówimy to w następnej sekcji, zależy od menedżera pamięci VTL 0, po to aby móc przydzielać i zwalniać pamięć VTL 1. Z kolei VTL 1 nie posiada żadnej pamięci fizycznej; na tym etapie polega na niektórych wcześniej przydzielonych (przez Windows Loader) stronach fizycznych, aby móc zaspokoić żądania przydzielenia pamięci. Gdy jądro NT startuje później, bezpieczne jądro wykonuje normalne wywołania żądania usług pamięci do menedżera pamięci VTL 0. W rezultacie niektóre części inicjalizacji bezpiecznego jądra muszą być odroczone po uruchomieniu jądra NT. Przepływ wykonania wraca do Loadera Windows w VTL 0. Ten ostatni ładuje i uruchamia jądro NT. Ostatnia część inicjalizacji bezpiecznego jądra zachodzi w fazie 0 i fazie 1 inicjalizacji jądra NT (więcej szczegółów w rozdziale 12.).

Faza 0 inicjalizacji jądra NT nadal nie ma dostępnych usług obsługi pamięci, ale jest to ostatni moment, w którym bezpieczne jądro w pełni ufa normalnemu światu. Sterowniki ładowane przez system wciąż nie zostały zainicjalizowane, a początkowy proces uruchamiania powinien być już chroniony przez bezpieczny rozruch (ang. *Secure Boot*). Bezpieczny handler wywołania `PHASE3_INIT` modyfikuje zabezpieczenia SLAT wszystkich fizycznych stron należących do bezpiecznego jądra i zależnych od niego modułów, czyniąc je niedostępnymi dla VTL 0. Ponadto stosuje ochronę typu tylko do odczytu dla bitmap CFG jądra. Na tym etapie bezpieczne jądro włącza obsługę integralności plików stron, tworzy początkowy proces systemowy i jego przestrzeń adresową oraz zapisuje wszystkie „zaufane" wartości współdzielonych rejestrów CPU (jak IDT, GDT, Syscall MSR, itd.). Struktury danych, na które wskazują współdzielone rejestry, są weryfikowane (dzięki bazie danych NTE). Na koniec uruchamiana jest bezpieczna pula wątków i inicjalizowany jest menedżer obiektów, moduł bezpiecznej integralności kodu (*Skci.dll*) oraz HyperGuard (więcej szczegółów na temat HyperGuard można znaleźć w rozdziale 7. w części I).

Gdy przepływ wykonawczy powraca do VTL 0, jądro NT może następnie uruchomić wszystkie pozostałe procesory aplikacji (AP — ang. *Application Processors*). Kiedy włączone jest bezpieczne jądro, inicjalizacja procesora aplikacji AP odbywa się w nieco inny sposób (inicjalizację procesora aplikacji AP omawiamy w następnym rozdziale).

W ramach fazy 1 inicjalizacji jądra NT system uruchamia menedżera we/wy. Menedżer we/wy, jak omówiono w części I, w rozdziale 6., „System operacji wejścia-wyjścia", jest rdzeniem systemu we/wy i definiuje model, w ramach którego żądania we/wy dostarczane są do sterowników urządzeń. Jednym z zadań menadżera we/wy jest inicjalizacja i uruchamianie sterowników rozruchowych oraz sterowników ELAM. Przed utworzeniem specjalnych sekcji do mapowania systemowych bibliotek DLL trybu użytkownika funkcja inicjalizacji menedżera we/wy emituje bezpieczne wywołanie `PHASE4_INIT`, aby rozpocząć ostatnią fazę inicjalizacji bezpiecznego jądra. Na tym etapie bezpieczne jądro nie ufa już VTL 0, ale może korzystać z usług świadczonych przez menedżera pamięci NT. Bezpieczne jądro inicjalizuje zawartość strony danych Współdzielonego bezpiecznego użytkownika (ang. *Secure User Shared*), która jest mapowana zarówno w trybie użytkownika VTL 1, jak i w trybie jądra, i kończy inicjalizację podsystemu wykonawczego. Odzyskuje wszystkie zasoby, które zostały zarezerwowane podczas procesu startowego, wywołuje każdy ze swoich punktów wejścia modułów zależnych (w szczególności *cng.sys* i *vmsvcext.sys*, które uruchamiają się przed wszelkimi normalnymi sterownikami startowymi).

Przydziela niezbędne zasoby do szyfrowania plików hibernacji, plików zrzutów pamięci przy awarii systemu (ang. *crash-dump*), plików stronicowania i integralności pamięci. Na koniec odczytuje i mapuje plik schematu zestawu API w pamięci VTL 1. Na tym etapie VSM jest całkowicie zainicjalizowany.

Uruchamianie procesorów aplikacji (AP)

Jednym z zabezpieczeń zapewnianych przez bezpieczne jądro jest uruchamianie procesorów aplikacji (AP), czyli takich, które nie są wykorzystywane do rozruchu systemu. Podczas startu systemu specyfikacje Intel i AMD architektur x86 i AMD64 definiują precyzyjny algorytm, który wybiera procesor rozruchowy (BSP — ang. *boot strap processor*) w systemach wieloprocesorowych. Procesor rozruchowy zawsze startuje w 16-bitowym trybie rzeczywistym (w którym ma dostęp do zaledwie 1 MB pamięci fizycznej) i zwykle wykonuje kod oprogramowania układowego firmware maszyny (w większości przypadków UEFI), który musi znajdować się w określonej lokalizacji pamięci fizycznej (lokalizacja ta jest nazywana wektorem resetu). Procesor rozruchowy wykonuje prawie całą inicjalizację systemu operacyjnego, hipernadzorcy i bezpiecznego jądra. Aby uruchomić pozostałe procesory niebędące procesorami rozruchowymi, system musi wysłać specjalne przerwanie międzyprocesorowe (IPI — ang. *inter-processor interrupt*) do lokalnych kontrolerów APIC należących do każdego procesora z osobna. Wektor startowy IPI (SIPI) zawiera adres pamięci fizycznej bloku startowego procesora, czyli bloku kodu zawierającego instrukcje umożliwiające wykonanie następujących podstawowych operacji:

1. Załaduj GDT i przełącz z 16-bitowego trybu rzeczywistego na 32-bitowy tryb chroniony (bez włączonego stronicowania).
2. Ustaw podstawową tablicę stron, włącz stronicowanie i wejdź w 64-bitowy tryb długi.
3. Załaduj 64-bitowe tablice deskryptorów przerwań IDT i tablice deskryptorów globalnych GDT, ustaw odpowiednie rejestry procesora i skocz do funkcji startowej systemu operacyjnego (*KiSystemStartup*).

Proces ten jest podatny na złośliwe ataki. Kod startowy procesora mógłby zostać zmodyfikowany przez zewnętrzne podmioty w trakcie jego wykonywania na procesorze aplikacji AP (jądro NT nie ma w tym momencie żadnej kontroli). W tym przypadku łatwo można obejść wszystkie obietnice bezpieczeństwa składane przez VSM. Gdy hipernadzorca i bezpieczne jądro są włączone, procesory aplikacji są nadal uruchamiane przez jądro NT, ale z wykorzystaniem hipernadzorcy.

KeStartAllProcessors, która jest funkcją wywoływaną przez fazę 1 inicjalizacji jądra NT (więcej szczegółów w rozdziale 12.), mając na celu uruchomienie wszystkich procesorów aplikacji AP, buduje współdzieloną tablicę IDT i wylicza wszystkie dostępne procesory, konsultując tablicę ACPI tablicy MADT (ang. *Multiple APIC Description Table*). Dla każdego odkrytego procesora funkcja ta przydziela pamięć dla bloku sterującego procesora (PRCB) i wszystkich prywatnych struktur danych procesora dla jądra i stosu DPC. Jeśli tryb VSM jest włączony, to następnie uruchamia procesor aplikacji AP, wysyłając bezpieczne wywołanie `START_PROCESSOR` do bezpiecznego jądra. Bezpieczne jądro sprawdza, czy wszystkie struktury danych przydzielone i wypełnione dla nowego procesora są ważne, w tym wartości początkowe rejestrów procesora i procedury rozruchowej (*KiSystemStartup*), i zapewnia, że rozruchy procesora aplikacji AP odbywają się sekwencyjnie i tylko raz na każdy procesor. Następnie inicjalizuje struktury danych VTL 1 potrzebne dla nowego procesora aplikacji (w szczególności SKPRCB). Uruchamiany jest wątek PRCB, który służy do wysyłania bezpiecznych wywołań w kontekście nowego procesora, a struktury danych VTL 0 procesora (CPU) są chronione za pomocą SLAT. Bezpieczne jądro włącza w końcu VTL 1 dla nowego procesora aplikacyjnego i uruchamia go za pomocą hiperwywołania *HvStartVirtualProcessor*.

Hipernadzorca uruchamia procesor aplikacji AP w podobny sposób, opisany na początku tego rozdziału (poprzez wysłanie startowego przerwania międzyprocesorowego IPI). W tym przypadku jednak procesor aplikacji AP rozpoczyna swoje wykonywanie w kontekście hipernadzorcy, przełącza się na wykonywanie w 64-bitowym trybie długim i wraca do VTL 1.

Pierwsza funkcja wykonywana przez procesor aplikacji rezyduje w VTL 1. Procedura inicjalizacji procesora bezpiecznego jądra mapuje stronę asysty procesora wirtualnego tworzoną na każdy procesor i stronę kontrolną SynIC, konfiguruje MBEC i włącza VINA. Następnie powraca do VTL 0 poprzez hiperwywołanie *HvVtlReturn*. Pierwszą procedurą wykonywaną w VTL 0 jest *KiSystemStartup*, która inicjalizuje struktury danych potrzebne jądru NT do zarządzania procesorem aplikacji AP, inicjuje HAL i przeskakuje do pętli bezczynności (więcej szczegółów w rozdziale 12.). Pętla rozsyłania bezpiecznego wywołania jest inicjalizowana później przez normalne jądro NT, gdy wykonywane jest pierwsze bezpieczne wywołanie.

Atakujący w tym przypadku nie może zmodyfikować bloku startowego procesora ani żadnej wartości początkowej jego rejestrów i struktur danych. W przypadku opisanego bezpiecznego uruchamiania procesora aplikacji AP, każda modyfikacja zostałaby wykryta przez bezpieczne jądro i system sprawdziłby błędy, aby pokonać wszelkie próby ataku.

Menedżer pamięci bezpiecznego jądra

Menedżer pamięci bezpiecznego jądra w dużym stopniu zależy od menedżera pamięci NT (oraz od menedżera pamięci programu Windows Loader z powodu jego kodu startowego). Całkowite opisanie menedżera pamięci bezpiecznego jądra wykracza poza zakres tej książki. Tutaj omówimy tylko najważniejsze koncepcje i struktury danych używane przez bezpieczne jądro.

Jak wspomniano w poprzednim rozdziale, inicjalizacja menedżera pamięci bezpiecznego jądra jest podzielona na trzy fazy. W fazie 1, najważniejszej, menedżer pamięci wykonuje następujące czynności:

1. Mapuje listę deskryptorów pamięci oprogramowania układowego firmware bootloadera w VTL 1, skanuje listę i określa pierwszą stronę fizyczną, której może użyć do przydzielenia pamięci potrzebnej do pierwszego uruchomienia (ten typ pamięci nazywa się SLAB). Mapuje tablice stron VTL 0 w adresie wirtualnym, który znajduje się dokładnie 512 GB przed tablicą stron VTL 1. Dzięki temu bezpieczne jądro może dokonać szybkiej konwersji pomiędzy adresem wirtualnym NT a tym pochodzącym z bezpiecznego jądra.
2. Inicjalizuje struktury danych zakresu wpisów PTE. Zakres wpisów w tablicy stron (PTE) zawiera bitmapę, która opisuje każdy kawałek przydzielonego zakresu adresów wirtualnych i pomaga bezpiecznemu jądru w przydzielaniu wpisów w tablicy stron (PTE) dla jego własnej przestrzeni adresowej.
3. Tworzy bazę bezpiecznych numerów ramek stron (ang. *Secure PFN*) i inicjalizuje pulę pamięci.
4. Inicjalizuje oszczędną (ang. *sparse*) tablicę adresową NT. Dla każdego załadowanego sterownika tworzy i wypełnia normalny zakres adresów (NAR — ang. *normal address range*), weryfikuje integralność binarną, wypełnia informacje o gorących poprawkach i — jeśli HVCI jest włączone — chroni każdą wykonywalną sekcję sterownika, używając SLAT. Następnie wykonuje cykle pomiędzy każdym wpisem PTE obrazu pamięci i zapisuje wpis tablicy adresowej NT (NTE — ang. *NT Address Table Entry*) w tablicy adresowej NT.
5. Inicjalizuje pakiety stron.

Bezpieczne jądro śledzi pamięć, z której korzysta normalne jądro NT. Menedżer pamięci bezpiecznego jądra używa struktury danych normalnego zakresu adresów NAR do opisania zakresu adresów wirtualnych jądra, które zawiera kod wykonywalny. NAR zawiera pewne informacje o zakresie (takie jak jego adres bazowy i rozmiar) oraz wskaźnik do struktury danych *SECURE_IMAGE*, która służy do opisywania sterowników uruchomieniowych (ogólnie rzecz biorąc, obrazów zweryfikowanych przy użyciu Secure HVCI, w tym obrazów trybu użytkownika używanych dla trustletów) ładowanych do VTL 0. Sterowniki ładowane w trybie rozruchowym (ang. *boot-loaded drivers*) nie korzystają ze struktury danych *SECURE_IMAGE*, ponieważ są one traktowane przez menedżera pamięci NT jako strony prywatne, które zawierają kod wykonywalny. Ta ostatnia struktura danych zawiera informacje dotyczące załadowanego obrazu w jądrze NT (które są weryfikowane przez bezpieczną integralność kodu jądra SKCI), takie jak adres jego punktu wejścia, kopię jego tablic relokacji (wykorzystywanych również do obsługi mechanizmu Retpoline i optymalizacji importu), wskaźnik do jego współdzielonych prototypowych wpisów PTE, informacje o łataniu „na gorąco” oraz strukturę danych określającą autoryzowane użycie jego stron pamięci. Struktura danych *SECURE_IMAGE* jest bardzo ważna, ponieważ jest używana przez bezpieczne jądro do śledzenia i weryfikacji współdzielonych stron pamięci, które są używane przez sterowniki uruchomieniowe.

Do śledzenia prywatnych stron jądra VTL 0 bezpieczne jądro używa struktury danych NTE. NTE istnieje dla każdej strony wirtualnej w przestrzeni adresowej VTL 0, która wymaga nadzoru ze strony bezpiecznego jądra; jest często używana w przypadku stron prywatnych. NTE śledzi wpisy PTE strony wirtualnej VTL 0 i przechowuje stan strony oraz informacje o jej ochronie. Gdy mechanizm HVCI jest włączony, tablica NTE dzieli wszystkie strony wirtualne na uprzywilejowane i nieuprzywilejowane. Strona uprzywilejowana odzwierciedla stronę pamięci, której jądro NT nie może samodzielnie „dotknąć”, ponieważ jest chroniona przez SLAT i zwykle odpowiada stronie wykonywalnej lub stronie CFG jądra przeznaczonej tylko do odczytu. Strona nieuprzywilejowana odzwierciedla wszystkie inne typy stron pamięci, nad którymi jądro NT ma pełną kontrolę. Bezpieczne jądro używa nieważnych wpisów NTE do odzwierciedlania stron nieuprzywilejowanych. Gdy mechanizm HVCI jest wyłączony, wszystkie strony prywatne są nieuprzywilejowane (jądro NT ma rzeczywiście pełną kontrolę nad wszystkimi swoimi stronami).

W systemach z włączonym mechanizmem HVCI menedżer pamięci NT nie może modyfikować żadnych chronionych stron. W przeciwnym razie w hipernadzorcy podniesie się wyjątek naruszenia rozszerzonych tablic stron EPT, co spowoduje awarię systemu. Po zakończeniu fazy rozruchowej tych systemów bezpieczne jądro przetworzyło już wszystkie niewykonywalne strony fizyczne poprzez SLAT — chroniąc je poprzez włączenie uprawnień dostępowych: odczyt i zapis. W tym scenariuszu nowe strony wykonywalne mogą być przydzielone tylko wtedy, gdy kod docelowy został zweryfikowany przez bezpieczny mechanizm HVCI.

Kiedy system, aplikacja lub menedżer Plug and Play wymagają załadowania nowego sterownika uruchomieniowego, rozpoczyna się złożona procedura angażująca NT i menedżera pamięci bezpiecznego jądra, którą w skrócie można podsumować następująco:

1. Menedżer pamięci NT tworzy obiekt sekcji, przydziela i wypełnia nowy obszar kontrolny (więcej szczegółów na temat menedżera pamięci NT jest dostępnych w rozdziale 5. w części I), odczytuje pierwszą stronę binariów i wywołuje bezpieczne jądro, mając na celu utworzenie względnego bezpiecznego obrazu, który opisuje nowo załadowany moduł.
2. Bezpieczne jądro tworzy strukturę danych *SECURE_IMAGE*, parsuje wszystkie sekcje pliku binarnego i wypełnia tablicę wpisów (PTE) bezpiecznych prototypów.

3. Jądro NT odczytuje cały plik binarny w niewykonywalnej pamięci współdzielonej (wskazanej przez prototypowe wpisy PTE obszaru kontrolnego). Wywołuje bezpieczne jądro, które korzystając z bezpiecznego mechanizmu HVCI, cyklicznie przechodzi między każdą sekcją obrazu binarnego i oblicza końcowy hash obrazu.
4. Jeśli obliczony hash pliku odpowiada hashowi zapisanemu w podpisie cyfrowym, pamięć NT przechodzi przez cały obraz i dla każdej strony wywołuje bezpieczne jądro, które sprawdza poprawność strony (hash każdej strony został już obliczony w poprzedniej fazie), stosuje potrzebne relokacje (randomizacja rozkładu przestrzeni adresowej (ASLR — ang. *Address Space Layout Randomization*), mechanizm Retpoline i optymalizacja importu) i stosuje nową ochronę translacji adresów drugiego poziomu SLAT, pozwalając stronie być wykonywalną, ale już nie zapisywalną.
5. Obiekt *Section* zostaje utworzony. Menedżer pamięci NT musi zmapować sterownik w swojej przestrzeni adresowej. Wywołuje on bezpieczne jądro w celu przydzielenia potrzebnych uprzywilejowanych wpisów PTE do opisania wirtualnego zakresu adresów sterownika. Bezpieczne jądro tworzy strukturę danych normalnego zakresu adresów NAR. Następnie mapuje strony fizyczne sterownika, które zostały wcześniej zweryfikowane, za pomocą procedury *MiMapSystemImage*.

Uwaga. Podczas inicjalizacji struktur danych NAR dla sterownika uruchomieniowego część tablicy jest wypełniana normalnymi wpisami NTE do opisu nowej przestrzeni adresowej sterownika. NTE nie są używane do śledzenia wirtualnego zakresu adresów sterownika uruchomieniowego (runtime) (jego strony wirtualne są współdzielone, a nie prywatne), więc względna część tablicy adresów NT jest wypełniona nieważnymi „zarezerwowanymi" NTE.

Podczas gdy zakresy adresów wirtualnych jądra VTL 0 są odzwierciedlane za pomocą struktury danych NAR, bezpieczne jądro używa bezpiecznych deskryptorów adresów wirtualnych (VAD — ang. *virtual address descriptor*) do śledzenia adresów wirtualnych trybu użytkownika w VTL 1. Bezpieczne VAD-y są tworzone za każdym razem, gdy tworzone jest nowe prywatne przydzielenie wirtualne, obraz binarny jest mapowany w przestrzeni adresowej trustletu (bezpiecznego procesu) oraz gdy tworzona jest enklawa VBS lub moduł jest mapowany w jej przestrzeni adresowej. Bezpieczny deskryptor adresu wirtualnego jest podobny do deskryptora VAD jądra NT i zawiera deskryptor zakresu VA, licznik referencji, kilka flag i wskaźnik do sekcji bezpiecznej (*Secure*), która została utworzona przez SKCI. (Wskaźnik do sekcji bezpiecznej (*Secure*) jest ustawiony na 0 w przypadku bezpiecznych VAD-ów opisujących prywatne przydzielenia (alokacje) wirtualne). Więcej szczegółów na temat trustletów i enklaw opartych na mechanizmie VBS zostanie omówionych w dalszej części tego rozdziału.

Identyfikacja stron i baza danych bezpiecznych numerów ramek stron PFN

Po załadowaniu sterownika i jego prawidłowym odwzorowaniu w pamięci VTL 0 menedżer pamięci NT musi być w stanie zarządzać jego stronami pamięci (z różnych powodów, takich jak stronicowanie sekcji sterownika o możliwościach stronicowania, tworzenie stron prywatnych, stosowanie prywatnych poprawek i tak dalej; więcej szczegółów w rozdziale 5. w części I). Za każdym razem, gdy menedżer pamięci NT operuje na pamięci chronionej, potrzebuje współpracy z bezpiecznym jądrem. Menedżerowi pamięci NT oferowane są dwa główne rodzaje bezpiecznych usług przy operowaniu na pamięci uprzywilejowanej: kopiowanie stron chronionych i usuwanie stron chronionych.

Struktura danych *PAGE_IDENTITY* jest spoiwem, które pozwala bezpiecznemu jądru śledzić wszystkie rodzaje stron. Struktura danych składa się z dwóch pól: kontekstu adresu i adresu wirtualnego. Za każdym razem, gdy jądro NT wywołuje bezpieczne jądro do pracy na stronach uprzywilejowanych, musi podać numer strony fizycznej wraz z poprawną strukturą danych *PAGE_IDENTITY*, opisującą, do czego służy strona fizyczna. Poprzez tę strukturę danych bezpieczne jądro może zweryfikować wykorzystanie żądanej strony i zdecydować, czy zezwolić na żądanie.

Tabela 9.4 pokazuje strukturę danych *PAGE_IDENTITY* (druga i trzecia kolumna) oraz wszystkie typy weryfikacji wykonywane przez bezpieczne jądro na różnych stronach pamięci:

- Jeśli bezpieczne jądro otrzyma żądanie skopiowania lub zwolnienia współdzielonej strony wykonywalnej sterownika uruchomieniowego, to sprawdza poprawność uchwytu bezpiecznego obrazu (określonego przez wywołującego) i uzyskuje jego odpowiednią strukturę danych (*SECURE_IMAGE*). Następnie wykorzystuje odpowiedni adres wirtualny (RVA) jako indeks do tablicy bezpiecznych prototypów, by uzyskać fizyczną ramkę strony (PFN) współdzielonej strony sterownika. Jeśli znaleziony numer ramki strony PFN jest równy podanemu przez wywołującego, bezpieczne jądro zezwala na żądanie; w przeciwnym razie blokuje je.
- W podobny sposób, jeśli jądro NT żąda operacji na stronie trustletu lub enklawy (więcej szczegółów na temat trustletów i bezpiecznych enklaw znajduje się w dalszej części tego rozdziału), bezpieczne jądro używa podanego przez wywołującego adresu wirtualnego do sprawdzenia, czy bezpieczny wpis PTE w tablicy bezpiecznych stron procesu zawiera prawidłowy numer PFN.
- Jak opisano wcześniej w punkcie „Menedżer pamięci bezpiecznego jądra”, w przypadku prywatnych stron jądra bezpieczne jądro lokalizuje wpis NTE zaczynający się od adresu wirtualnego określonego przez wywołującego i sprawdza, czy zawiera on poprawny numer PFN, który musi być taki sam jak określony przez wywołującego.
- Strony znaczników miejsc (ang. *placeholder*) to wolne strony, które są chronione przez SLAT. Bezpieczne jądro weryfikuje stan strony znaczników miejsc za pomocą bazy danych numerów ramek stron PFN.

TABELA 9.4. *Różne tożsamości stron zarządzane przez bezpieczne jądro*

Typ strony	Kontekst adresu	Adres wirtualny	Struktura weryfikacyjna
Współdzielona przez jądro	Bezpieczny uchwyt obrazu	RVA strony	Bezpieczny prototyp PTE
Trustlet/enklawa	Bezpieczny uchwyt procesu	Adres wirtualny bezpiecznego procesu	Bezpieczny wpis PTE
Prywatne jądro	0	Adres wirtualny jądra strony	Wpis do tablicy adresowej NT (NTE)
Znacznik miejsca	0	0	Wpis PFN

Menedżer pamięci bezpiecznego jądra utrzymuje bazę numerów ramek stron PFN, aby odzwierciedlać stan każdej strony fizycznej. Wpis PFN w bezpiecznym jądrze jest znacznie mniejszy w porównaniu z jego odpowiednikiem w NT; zasadniczo zawiera on stan strony i licznik udziałów. Z perspektywy bezpiecznego jądra strona fizyczna może znajdować się w jednym z następujących stanów: nieważna, wolna, współdzielona, we/wy, bezpieczna lub obraz (zabezpieczonego prywatnego jądra NT).

Stan zabezpieczony jest używany dla stron fizycznych, które są prywatne dla bezpiecznego jądra (jądro NT nigdy nie może się o nie upomnieć), lub dla stron fizycznych, które zostały przydzielone przez jądro NT, a następnie zabezpieczone przez bezpieczne jądro przy pomocy translacji SLAT do przechowywania kodu wykonywalnego zweryfikowanego przez bezpieczny mechanizm HVCI. Tylko zabezpieczone nieprywatne strony fizyczne mają tożsamość strony.

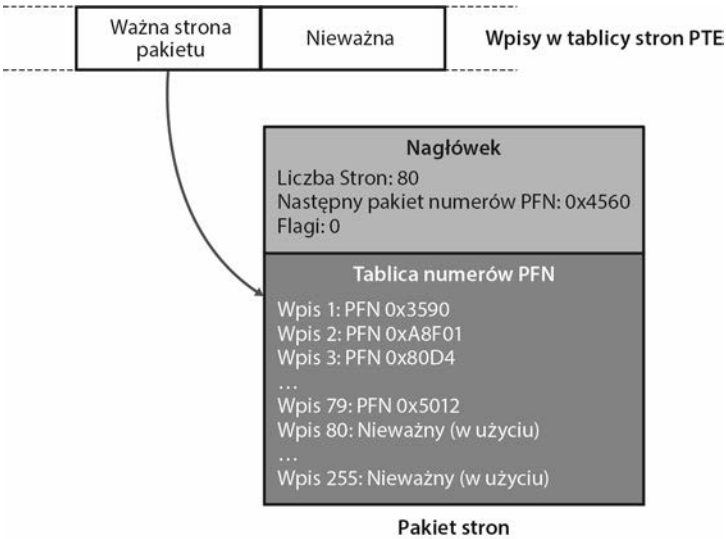
Gdy jądro NT zamierza wyrzucić zabezpieczoną stronę, prosi bezpieczne jądro o operację usunięcia strony. Bezpieczne jądro analizuje określoną tożsamość strony i dokonuje jej weryfikacji (jak wyjaśniono to wcześniej). W przypadku gdy tożsamość strony odnosi się do enklawy lub strony trustletu, bezpieczne jądro szyfruje zawartość strony przed wydaniem jej jądru NT, które następnie przechowuje stronę w pliku stronicowania. W ten sposób jądro NT nadal nie ma szans na przechwycenie prawdziwej zawartości pamięci prywatnej.

Przydzielanie bezpiecznej pamięci

Jak omówiono to w poprzednich rozdziałach, gdy bezpieczne jądro rozpoczyna pracę, analizuje listy deskryptorów pamięci oprogramowania układowego firmware, aby móc przydzielić pamięć fizyczną na własny użytek. W fazie pierwszej swojej inicjalizacji bezpieczne jądro nie może korzystać z usług pamięciowych dostarczanych przez jądro NT (jądro NT rzeczywiście nie jest jeszcze zainicjalizowane), więc używa wolnych wpisów na listach deskryptorów pamięci oprogramowania układowego firmware do rezerwowania dwumegabajtowych SLAB-ów. SLAB to 2 MB ciągłej pamięci fizycznej, która jest mapowana przez pojedynczy zagnieżdżony wpis katalogu tablicy stron w hipernadzorcy. Wszystkie strony pamięci SLAB mają taką samą ochronę SLAT. Pamięci SLAB zostały zaprojektowane z myślą o wydajności. Dzięki mapowaniu dwumegabajtowego kawałka pamięci fizycznej za pomocą pojedynczego wpisu tablicy stron zagnieżdżonych w hipernadzorcy dodatkowa sprzętowa translacja adresów pamięci jest szybsza i skutkuje mniejszą liczbą pominięć pamięci podręcznej w tablicy SLAT.

Pierwszy *pakiet stron* bezpiecznego jądra jest wypełniony 1 MB przydzielonej pamięci SLAB. Pakiet stron to struktura danych pokazana na rysunku 9.37, która zawiera listę ciągłych wolnych numerów ramek stron fizycznych PFN. Kiedy bezpieczne jądro potrzebuje pamięci do własnych celów, przydziela strony fizyczne z pakietu stron, usuwając jedną lub większą ilość wolnych ramek stron z końca tablicy PFN pakietu. W tym przypadku bezpieczne jądro nie musi sprawdzać listy deskryptorów pamięci oprogramowania układowego firmware, aż do momentu gdy pakiet zostanie całkowicie zużyty. Po zakończeniu fazy trzeciej inicjalizacji bezpiecznego jądra usługi pamięciowe jądra NT stają się dostępne, a więc bezpieczne jądro zwalnia wszelkie listy deskryptorów pamięci rozruchowej, zachowując strony pamięci fizycznej znajdujące się wcześniej w pakietach.

Przyszłe bezpieczne przydziały pamięci używają normalnych wywołań dostarczanych przez jądro NT. Pakiety stron zostały zaprojektowane tak, aby zminimalizować liczbę normalnych wywołań potrzebnych do przydzielenia pamięci. Gdy pakiet zostanie w pełni przydzielony, nie zawiera żadnych stron (wszystkie jego strony są obecnie przydzielone), a nowy zostanie wygenerowany przez poproszenie jądra NT o 1 MB ciągłych stron fizycznych (poprzez normalne wywołanie *ALLOC_PHYSICAL_PAGES*). Pamięć fizyczna zostanie przydzielona przez jądro NT z właściwego SLAB-a.



RYSUNEK 9.37. *Pakiet bezpiecznych stron z 80 dostępnymi stronami. Pakiet składa się z nagłówka i tablicy wolnych numerów PFN*

W ten sam sposób za każdym razem, gdy bezpieczne jądro zwalnia część swojej prywatnej pamięci, przechowuje odpowiadające jej strony fizyczne we właściwym pakiecie, powiększając swoją tablicę PFN do limitu 256 wolnych stron. Gdy tablica zostanie całkowicie wypełniona, a pakiet stanie się wolny, nowy element roboczy jest umieszczany w kolejce. Ten element roboczy wyzeruje wszystkie strony i wyemituje normalne wywołanie *FREE_PHYSICAL_PAGES*, które zakończy się wykonaniem funkcji *MmFreePagesFromMdl* menedżera pamięci NT.

Za każdym razem, gdy wystarczająca ilość stron jest przenoszona do i z pakietu, są one w pełni chronione w VTL 0 przez użycie translacji SLAT (ta procedura jest nazywana „zabezpieczeniem pakietu"). Bezpieczne jądro obsługuje trzy rodzaje pakietów, z których wszystkie przydzielają pamięć z różnych SLAB-ów: Brak dostępu, tylko odczyt, odczyt-wykonanie.

Łatanie „na gorąco"

Kilka lat temu 32-bitowe wersje Windows obsługiwały „gorące" łaty (ang. *hot patch*) składników systemu operacyjnego. Funkcje, które mogły być łatane, zawierały nadmiarowy dwubajtowy kod operacyjny (*opcode*) w swoim prologu oraz kilka bajtów wypełnienia (*padding*) znajdujących się przed samą funkcją. Pozwalało to jądru NT na dynamiczne zastąpienie początkowego kodu operacyjnego skokiem pośrednim, wykorzystującym wolną przestrzeń zapewnioną przez wypełnienie, w celu przekierowania kodu do łatanej funkcji rezydującej w innym module. Funkcja ta była mocno wykorzystywana przez Windows Update, co pozwalało na aktualizację systemu bez konieczności natychmiastowego restartu maszyny. Przy przejściu na architektury 64-bitowe nie było to już możliwe ze względu na różne problemy. Dobrym przykładem była ochrona przed łatami jądra; nie było już niezawodnego sposobu na modyfikację chronionych binariów trybu jądra i na pozwolenie strażnikowi łat (PatchGuard) na aktualizacje bez narażania niektórych jego prywatnych interfejsów, a odsłonięte interfejsy strażnika łat mogły być łatwo wykorzystane przez atakującego, którego celem było pokonanie ochrony.

Bezpieczne jądro rozwiązało wszystkie problemy związane z 64-bitowymi architekturami i ponownie wprowadziło do systemu operacyjnego możliwość łatania „na gorąco" binariów jądra. Kiedy bezpieczne jądro jest włączone, następujące typy obrazów wykonywalnych mogą być łatane „na gorąco":

- moduły trybu użytkownika VTL 0 (zarówno pliki wykonywalne, jak i biblioteki);
- sterowniki trybu jądra, HAL i binaria jądra NT, chronione lub nie przez strażnika łat (PatchGuard);
- binaria bezpiecznego jądra i jego moduły zależne, które działają w trybie jądra VTL 1;
- hipernadzorca (Intel, AMD i wersja ARM).

Binaria łat stworzone do ukierunkowania na oprogramowanie działające w VTL 0 są nazywane normalnymi łatami, natomiast pozostałe są nazywane bezpiecznymi łatami. Jeśli bezpieczne jądro nie jest włączone, można łatać tylko aplikacje trybu użytkownika.

Obraz „gorącej" łaty jest standardową binarną wersją przenośnego pliku wykonywalnego (PE — ang. *Portable Executable*), która zawiera tablicę „gorących" łat, strukturę danych używaną do śledzenia funkcji łat. Tablica „gorących" łat jest połączona w binariach poprzez katalog danych konfiguracyjnych ładowania obrazu. Zawiera ona jeden lub więcej deskryptorów opisujących każdy możliwy do załatania obraz bazowy, który jest identyfikowany przez jego sumę kontrolną i znacznik daty czasu. (W ten sposób „gorąca" łata jest kompatybilna tylko z właściwymi obrazami bazowymi. System nie może zastosować łaty do niewłaściwego obrazu). Tablica „gorących" łat zawiera również listę funkcji lub globalnych kawałków danych, które muszą być zaktualizowane w obrazie bazowym lub w obrazie łaty; silnik łat opiszemy wkrótce. Każdy wpis na tej liście zawiera przesunięcia funkcji w obrazie bazowym i obrazie łaty oraz oryginalne bajty funkcji bazowej, które zostaną zastąpione.

Do obrazu bazowego można zastosować wiele łat, ale stosowanie łat jest idempotentne. Ta sama łata może być stosowana wielokrotnie lub różne łaty mogą być stosowane w sekwencji. Niezależnie od tego ostatnio zastosowana łata będzie aktywną łatą dla obrazu bazowego. Kiedy system musi zastosować „gorącą" łatę, używa wywołania systemowego *NtManageHotPatch*, które jest używane do instalowania, usuwania i zarządzania gorącymi łatami. (Wywołanie systemowe obsługuje różne klasy „informacji o łatach" dla opisania wszystkich możliwych operacji). „Gorąca" łata może być zainstalowana globalnie dla całego systemu lub — jeśli łata dotyczy kodu trybu użytkownika (VTL 0) — dla wszystkich procesów należących do określonej sesji użytkownika.

Gdy system żąda zastosowania łaty, jądro NT lokalizuje tablicę „gorących" łat w binarnym pliku łaty i sprawdza jej poprawność. Następnie używa bezpiecznego wywołania *DETERMINE_HOT_PATCH_TYPE* do bezpiecznego określenia typu łaty. W przypadku bezpiecznej łaty tylko bezpieczne jądro może ją zastosować, więc używane jest bezpieczne wywołanie *APPLY_HOT_PATCH*; żadne inne przetwarzanie przez jądro NT nie jest potrzebne. We wszystkich innych przypadkach jądro NT najpierw próbuje zastosować łatę do sterownika jądra. Przechodzi cyklicznie między każdym załadowanym modułem jądra, szukając obrazu bazowego, który ma taką samą sumę kontrolną opisaną przez jeden z deskryptorów „gorącej" łaty obrazu.

Łatanie „na gorąco" jest włączone tylko wtedy, gdy wartość rejestru `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\HotPatchTableSize` jest wielokrotnością standardowego rozmiaru strony pamięci (4096). W rzeczywistości, gdy włączone jest łatanie „na gorąco", każdy obraz, który jest mapowany w wirtualnej przestrzeni adresowej, musi mieć zarezerwowaną pewną ilość wirtualnej przestrzeni adresowej zaraz po samym obrazie. Ta zarezerwowana przestrzeń jest używana dla tablicy adresowej gorących łat

(HPAT — ang. *hot patch address table*, nie mylić z tablicą „gorących" łat). HPAT jest używany do zminimalizowania ilości bajtów wypełnienia (*padding*) potrzebnych dla każdej funkcji, która ma być załatana, poprzez przechowywanie adresu nowej funkcji w załatanym obrazie. Podczas łatania funkcji lokalizacja tablicy HPAT zostanie użyta do wykonania pośredniego skoku z oryginalnej funkcji w obrazie bazowym do łatanej funkcji w obrazie łatanym (zauważ, że dla zgodności z Retpoline zamiast skoku pośredniego używany jest inny rodzaj procedury Retpoline).

Gdy jądro NT znajdzie sterownik trybu jądra odpowiedni dla łaty, ładuje i mapuje binarny obraz łaty w przestrzeni adresowej jądra i tworzy związany z nim wpis w tablicy danych programu ładującego (*loader*) (więcej szczegółów w rozdziale 12.). Następnie skanuje każdą stronę pamięci zarówno obrazu podstawowego, jak i obrazu łaty i blokuje w pamięci te strony, które są zaangażowane w łatanie „na gorąco" (jest to ważne; w ten sposób strony nie mogą być stronicowane na dysk, gdy trwa aplikowanie łaty). Na koniec emituje bezpieczne wywołanie *APPLY_HOT_PATCH*.

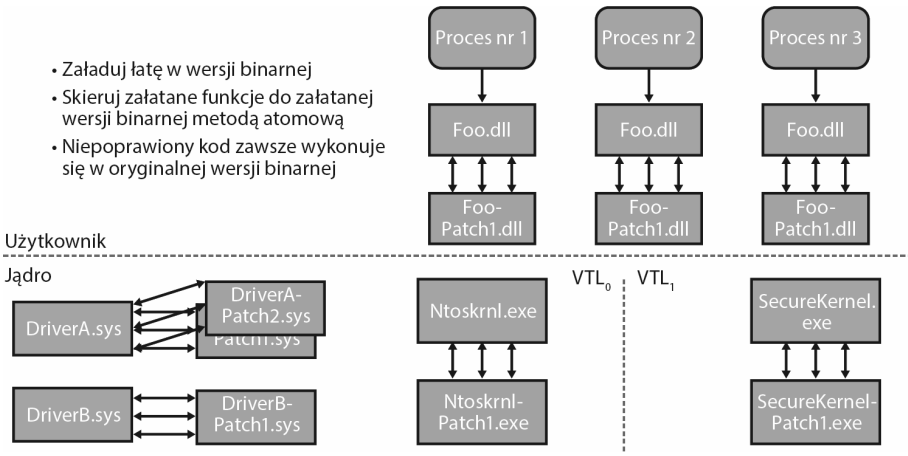
Prawdziwy proces aplikowania łaty rozpoczyna się w bezpiecznym jądrze. Przechwytuje ono i weryfikuje tablicę „gorących" łat obrazu łaty (poprzez ponowne zmapowanie obrazu łaty również w VTL 1) i lokalizuje NAR obrazu bazowego (zobacz poprzedni punkt, „Menedżer pamięci bezpiecznego jądra", aby uzyskać więcej szczegółów na temat NAR), który również mówi bezpiecznemu jądru, czy obraz jest chroniony przez strażnika łat. Następnie bezpieczne jądro sprawdza, czy w obrazie HPAT dostępna jest wystarczająca ilość zarezerwowanego miejsca. Jeśli tak, to przydziela jedną lub więcej wolnych stron fizycznych (pobierając je z bezpiecznego pakietu (ang. *secure bundle*) lub używając normalnego wywołania *ALLOC_PHYSICAL_PAGES*), które będą mapowane w zarezerwowanej przestrzeni. W tym momencie, jeśli obraz bazowy jest chroniony, bezpieczne jądro rozpoczyna złożony proces, który aktualizuje wewnętrzny stan strażnika łat dla nowego, załatanego obrazu i ostatecznie wywołuje silnik łatający.

Silnik łatający jądra wykonuje następujące operacje wysokiego poziomu, które są opisane przez inny typ wpisu w tablicy „gorących" łat:

1. Łata wszystkie wywołania z łatanych funkcji w obrazie łaty, mając na celu przeskoczenie do odpowiednich funkcji w obrazie bazowym. Dzięki temu cały niezałatany kod zawsze wykonuje się w oryginalnym obrazie bazowym. Na przykład, jeśli funkcja A wywołuje B w obrazie bazowym, a łata zmienia funkcję A, ale nie funkcję B, to silnik łaty zaktualizuje funkcję B w łacie, aby przeskoczyć do funkcji B w obrazie bazowym.
2. Łata niezbędne odniesienia do zmiennych globalnych w łatanych funkcjach, aby wskazywały na odpowiadające im zmienne globalne w obrazie bazowym.
3. Łata niezbędne odwołania do tablicy adresów importu (IAT — ang. *Import Address Table*) w obrazie łaty poprzez skopiowanie odpowiednich wpisów IAT z obrazu bazowego.
4. W sposób „atomowy" łata niezbędne funkcje w obrazie bazowym, aby przeskoczyć do odpowiedniej funkcji w obrazie łaty. Gdy tylko zostanie to zrobione dla danej funkcji w obrazie bazowym, wszystkie nowe wywołania tej funkcji wykonają nowy, załatany kod funkcji w obrazie łaty. Gdy załatana funkcja zostanie zwrócona, zwróci do wywołującego oryginalną funkcję w obrazie bazowym.

Ponieważ wskaźniki nowych funkcji mają szerokość 64 bitów (8 bajtów), silnik łaty wstawia każdy wskaźnik do HPAT, który znajduje się na końcu binariów. W ten sposób potrzebuje tylko 5 bajtów na umieszczenie skoku pośredniego w przestrzeni wypełnienia (padding) znajdującej się na początku każdej funkcji (proces został uproszczony. „Gorące" łaty kompatybilne z Retpoline wymagają kompatybilnego Retpoline. Ponadto HPAT jest podzielony na stronę kodu i stronę danych).

Jak widać na rysunku 9.38, silnik łat jest kompatybilny z różnymi rodzajami binariów. Jeśli jądro NT nie znalazło żadnego możliwego do załatania modułu trybu jądra, wznawia wyszukiwanie poprzez wszystkie procesy trybu użytkownika i stosuje procedurę podobną do prawidłowego łatania „na gorąco" zgodnego programu wykonywalnego lub biblioteki trybu użytkownika.



RYSUNEK 9.38. *Schemat działania silnika do łatania „na gorąco" na różnych typach binariów*

Tryb izolowanego użytkownika

Tryb izolowanego użytkownika (IUM — ang. *Isolated User Mode*), usługi dostarczane przez bezpieczne jądro do jego bezpiecznych procesów (trustletów) oraz ogólna architektura trustletów zostały omówione w rozdziale 3. części I. W tym rozdziale kontynuujemy dyskusję od tego miejsca i przechodzimy do opisu niektórych usług dostarczanych przez tryb izolowanego użytkownika, takich jak bezpieczne urządzenia i enklawy VBS.

Jak opisano to w rozdziale 3. części I, kiedy trustlet jest tworzony w VTL 1, zwykle mapuje w swojej przestrzeni adresowej następujące biblioteki:

- ***Iumdll.dll*** — Biblioteka DLL warstwy natywnej IUM implementuje bezpieczny stub wywołania systemowego. Jest to odpowiednik biblioteki *Ntdll.dll* od VTL 0.
- ***Iumbase.dll*** — Biblioteka DLL IUM warstwy bazowej jest biblioteką implementującą większość bezpiecznych API, które mogą być konsumowane wyłącznie przez oprogramowanie VTL 1. Zapewnia ona różne usługi dla każdego bezpiecznego procesu, takie jak bezpieczna identyfikacja, komunikacja, kryptografia i bezpieczne zarządzanie pamięcią. Trustlety zazwyczaj nie wywołują bezpośrednio bezpiecznych wywołań systemowych, ale przechodzą przez *Iumbase.dll*, który jest odpowiednikiem *kernelbase.dll* w VTL 0.
- ***IumCrypt.dll*** — Prezentuje funkcje szyfrowania klucza publicznego/prywatnego używane do podpisywania i weryfikacji integralności. Większość funkcji kryptograficznych prezentowanych w VTL 1 jest zaimplementowana w *Iumbase.dll*; tylko niewielka liczba specjalistycznych procedur szyfrowania jest zaimplementowana w IumCrypt. LsaIso jest głównym konsumentem usług prezentowanych przez IumCrypt, który nie jest ładowany w wielu innych trustletach.

- ***Ntdll.dll***, ***Kernelbase.dll*** i ***Kernel32.dll*** — Trustlet może być zaprojektowany do działania zarówno w VTL 1 jak i VTL 0. W takim przypadku powinien używać tylko procedur zaimplementowanych w standardowej powierzchni API VTL 0. Nie wszystkie usługi dostępne dla VTL 0 są również zaimplementowane w VTL 1. Na przykład trustlet nigdy nie może wykonywać żadnych operacji we/wy z rejestru ani żadnych operacji we/wy z plików, ale może używać procedur synchronizacji, ALPC, interfejsów API wątków i strukturalnej obsługi wyjątków, a także może zarządzać pamięcią wirtualną i obiektami sekcji. Prawie wszystkie usługi oferowane przez biblioteki *kernelbase* i *kernel32* wykonują wywołania systemowe poprzez *Ntdll.dll*. W VTL 1 tego rodzaju wywołania systemowe są „tłumaczone" na normalne wywołania i przekierowywane do jądra VTL 0. (Normalne wywołania omówiliśmy szczegółowo wcześniej w tym rozdziale). Normalne wywołania są często używane przez funkcje IUM i przez samo bezpieczne jądro. To wyjaśnia, dlaczego *ntdll.dll* jest zawsze mapowany w każdym trustlecie.
- ***Vertdll.dll*** — DLL wykonawczy enklawy VSM jest DLL-em, która zarządza czasem życia enklawy VBS. Oprogramowanie działające w bezpiecznej enklawie świadczy tylko ograniczone usługi. Biblioteka ta implementuje wszystkie usługi enklawy prezentowane oprogramowaniu enklawy i zwykle nie jest ładowana dla standardowych procesów VTL 1.

Mając na uwadze tę wiedzę, przyjrzyjmy się, co jest zaangażowane w proces tworzenia trustletów, zaczynając od API *CreateProcess* w VTL 0, dla którego przepływ jego wykonania został już szczegółowo opisany w rozdziale 3.

Tworzenie trustletów

Jak wielokrotnie omawiano w poprzednich rozdziałach, bezpieczne jądro zależy od jądra NT przy wykonywaniu różnych operacji. Tworzenie trustletów odbywa się według tej samej zasady. Jest to operacja zarządzana zarówno przez bezpieczne jądro, jak i jądro NT. W rozdziale 3. części I przedstawiliśmy strukturę trustletu i wymóg jego podpisywania, a także opisaliśmy ważne metadane dotyczące zasad. Ponadto opisaliśmy szczegółowy przepływ API *CreateProcess*, który nadal jest punktem wyjścia do tworzenia trustletów.

Aby poprawnie utworzyć trustlet, aplikacja powinna określić flagę tworzenia `CREATE_SECURE_PROCESS` podczas wywoływania API *CreateProcess*. Wewnętrznie flaga ta jest konwertowana na atrybut NT `PS_CP_SECURE_PROCESS` i przekazywana do natywnego API *NtCreateUserProcess*. Po tym jak *NtCreateUserProcess* pomyślnie otworzy obraz do wykonania, tworzy obiekt sekcji obrazu, określając specjalną flagę, która instruuje menedżera pamięci, aby użył Secure HVCI do sprawdzenia jego zawartości. Dzięki temu bezpieczne jądro tworzy strukturę danych *SECURE_IMAGE* używaną do opisu obrazu PE zweryfikowanego poprzez bezpieczną HVCI.

Jądro NT tworzy wymagane struktury danych procesu i początkową przestrzeń adresową VTL 0 (katalogi stron, hiperprzestrzeń i zbiór roboczy) jak dla normalnych procesów, a jeśli nowy proces jest trustletem, emituje bezpieczne wywołanie *CREATE_PROCESS*. Bezpieczne jądro zarządza tym ostatnim, tworząc bezpieczny obiekt procesu i odpowiednią strukturę danych (nazwaną *SEPROCESS*). Bezpieczne jądro łączy normalny obiekt procesu (*EPROCESS*) z nowym bezpiecznym obiektem i tworzy początkową bezpieczną przestrzeń adresową poprzez przydzielenie tablicy bezpiecznych stron i powielanie wpisów głównych opisujących część jądra bezpiecznej przestrzeni adresowej w jej górnej połowie.

Jądro NT kończy konfigurację pustej przestrzeni adresowej procesu i mapuje do niej bibliotekę Ntdll (więcej szczegółów w opisie etapu 3D w rozdziale 3. części I). Podczas wykonywania tego dla bezpiecznych procesów jądro NT wywołuje bezpieczne wywołanie *INITIALIZE_PROCESS*, by zakończyć konfigurację w VTL 1. Bezpieczne jądro kopiuje do nowego bezpiecznego procesu tożsamość trustletu i atrybuty trustletu określone podczas tworzenia procesu, tworzy bezpieczną tablicę uchwytów i mapuje bezpieczną współdzieloną stronę do przestrzeni adresowej.

Ostatnim krokiem potrzebnym dla bezpiecznego procesu jest utworzenie bezpiecznego wątku. Początkowy obiekt wątku tworzony jest tak samo jak w przypadku normalnych procesów w jądrze NT. Kiedy *NtCreateUserProcess* wywołuje *PspInsertThread*, przydzielił już stos jądra wątku i wstawił niezbędne dane do startu z funkcji jądra *KiStartUserThread* (więcej szczegółów w opisie etapu 4. w rozdziale 3. części I). Jeśli proces jest trustletem, jądro NT emituje bezpieczne wywołanie *CREATE_THREAD* w celu wykonania ostatecznego bezpiecznego utworzenia wątku. Bezpieczne jądro dołącza się do przestrzeni adresowej nowego bezpiecznego procesu, przydziela i inicjalizuje bezpieczną strukturę danych wątku, bezpieczny blok środowiskowy wątku (TEB — ang. *Thread Environment Block*) i stos jądra. Bezpieczne jądro wypełnia stos jądra wątku, wstawiając pierwszą procedurę początkową jądra wątku: *SkpUserThreadStart*. Następnie inicjalizuje zależny od maszyny kontekst sprzętowy dla bezpiecznego wątku, który określa rzeczywisty adres startowy obrazu i adres pierwszej procedury trybu użytkownika. Na koniec łączy normalny obiekt wątku z nowo utworzonym bezpiecznym, wstawia wątek do listy bezpiecznych wątków i oznacza go jako możliwy do uruchomienia.

Gdy obiekt normalnego wątku zostanie wybrany do uruchomienia przez planistę jądra NT, wykonanie rozpoczyna jeszcze funkcja *KiStartUserThread* w VTL 0. Ta ostatnia obniża IRQL wątku i wywołuje systemową procedurę początkową wątku (*PspUserThreadStartup*). Wykonanie przebiega jak dla zwykłych wątków, aż do momentu gdy jądro NT ustawi początkowy kontekst wątku. Zamiast tego uruchamia pętlę rozsyłania bezpiecznego jądra, wywołując procedurę *VslpEnterIumSecureMode* i określając bezpieczne wywołanie *RESUMETHREAD*. Pętla kończy się dopiero wtedy, gdy wątek zostanie zakończony. Początkowe bezpieczne wywołanie jest przetwarzane przez pętlę rozsyłania normalnych wywołań w VTL 1, która identyfikuje powód wejścia do VTL 1 jako „wznowienie wątku", dołącza do przestrzeni adresowej nowego procesu i przełącza się na nowy bezpieczny stos wątków. Bezpieczne jądro w tym przypadku nie wywołuje funkcji rozsyłania *IumInvokeSecureService*, ponieważ wie, że funkcja początkowa wątku znajduje się na stosie, więc po prostu wraca do adresu znajdującego się na stosie, który wskazuje na bezpieczną procedurę początkową VTL 1, *SkpUserThreadStart*.

SkpUserThreadStart, podobnie jak w przypadku standardowych wątków VTL 0, ustawia początkowy kontekst thunk, aby uruchomić procedurę inicjalizacyjną programu ładującego obraz (*LdrInitializeThunk* w *Ntdll.dll*), jak również systemowy stub startowy wątku (*RtlUserThreadStart* w *Ntdll.dll*). Kroki te są wykonywane poprzez edycję kontekstu wątku na miejscu, a następnie wyemitowanie operacji wyjścia z usługi systemowej, która ładuje specjalnie spreparowany kontekst użytkownika i powraca do trybu użytkownika. Inicjalizacja bezpiecznego nowo utworzonego wątku przebiega tak jak w przypadku zwykłych wątków VTL 0; procedura *LdrInitializeThunk* inicjalizuje program ładujący (Loader) i struktury danych, których Loader potrzebuje. Po powrocie funkcji *NtContinue* przywraca nowy kontekst użytkownika. Teraz naprawdę rozpoczyna się wykonywanie wątków. *RtlUserThreadStart* używa adresu rzeczywistego punktu wejścia obrazu oraz parametru start i wywołuje punkt wejścia aplikacji.

Uwaga. Uważny czytelnik mógł zauważyć, że bezpieczne jądro nie robi nic, aby chronić obraz binarny nowego trustletu. Dzieje się tak dlatego, że pamięć współdzielona, która opisuje podstawowy obraz binarny trustletu, jest nadal dostępna dla VTL 0.

Załóżmy, że trustlet chce zapisać prywatne dane znajdujące się w globalnych danych obrazu. PTE, które mapują sekcję danych zapisywalnych w globalnych danych obrazu, są oznaczone jako kopiowane przy zapisie (copy-on-write). W związku z tym procesor wygeneruje błąd dostępu. Błąd należy do zakresu adresów trybu użytkownika (pamiętaj, że do śledzenia stron współdzielonych nie są używane NAR). Obsługa błędu strony bezpiecznego jądra przekazuje wykonanie do jądra NT (poprzez normalne wywołanie), które przydzieli nową stronę, skopiuje na nią zawartość starej i zabezpieczy ją poprzez SLAT (używając operacji chronionego kopiowania; więcej szczegółów w punkcie „Menedżer pamięci bezpiecznego jądra" we wcześniejszej części tego rozdziału).

Eksperyment: debugowanie trustletu

Debugowanie trustletu za pomocą debugera trybu użytkownika jest możliwe tylko wtedy, gdy trustlet wyraźnie na to pozwala poprzez swoje metadane zasad (przechowywane w sekcji `.tPolicy`). W tym eksperymencie spróbujemy debugować trustlet poprzez debuger jądra. Potrzebujesz debugera jądra podłączonego do systemu testowego (lokalny debuger jądra też działa), który musi mieć włączony VBS. Integralność HVCI nie jest jednak bezwzględnie wymagana.

Najpierw znajdź trustlet *LsaIso.exe*:

```
lkd> !process 0 0 lsaiso.exe
PROCESS ffff8904dfdaa080
    SessionId: 0    Cid: 02e8    Peb: 8074164000    ParentCid: 0250
    DirBase: 3e590002    ObjectTable: ffffb00d0f4dab00    HandleCount: 42.
    Image: LsaIso.exe
```

Analiza bloku środowiska procesu (PEB — ang. *process environment block*) ujawnia, że niektóre informacje są ustawione na 0 lub są nieczytelne:

```
lkd> .process /P ffff8904dfdaa080
lkd> !peb 8074164000
PEB at 0000008074164000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00007ff708750000
    NtGlobalFlag:             0
    NtGlobalFlag2:            0
    Ldr                       0000000000000000
    *** unable to read Ldr table at 0000000000000000
    SubSystemData:     0000000000000000
    ProcessHeap:       0000000000000000
    ProcessParameters: 0000026b55a10000
    CurrentDirectory:  'C:\Windows\system32\'
    WindowTitle:   '< Name not readable >'
    ImageFile:     '\??\C:\Windows\system32\lsaiso.exe'
    CommandLine:   '\??\C:\Windows\system32\lsaiso.exe'
    DllPath:    '  < Name not readable >'lkd
```

Odczyt z adresu bazowego obrazu procesu może się udać, ale zależy to od tego, czy obraz LsaIso odwzorowany w przestrzeni adresowej VTL 0 został już udostępniony. Zwykle dotyczy to tylko pierwszej strony (pamiętaj, że pamięć współdzielona głównego obrazu jest dostępna w VTL 0). W naszym systemie pierwsza strona jest zmapowana i ważna, natomiast trzecia jest nieważna:

```
lkd> db 0x7ff708750000 l20
00007ff7`08750000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff 00 00  MZ.............
00007ff7`08750010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00 ........@.......
lkd> db (0x7ff708750000 + 2000) l20
00007ff7`08752000 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
00007ff7`08752010 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
lkd> !pte (0x7ff708750000 + 2000)
1: kd> !pte (0x7ff708750000 + 2000)
                                            VA 00007ff708752000
PXE at FFFFD5EAF57AB7F8    PPE at FFFFD5EAF56FFEE0    PDE at FFFFD5EADFFDC218
contains 0A0000003E58D867  contains 0A0000003E58E867  contains 0A0000003E58F867
pfn 3e58d     ---DA--UWEV  pfn 3e58e     ---DA--UWEV  pfn 3e58f     ---DA--UWEV

PTE at FFFFD5BFFB843A90
contains 0000000000000
not valid
```

Dumping wątków procesu ujawnia ważne informacje, które potwierdzają to, co omawialiśmy w poprzednich rozdziałach:

```
!process ffff8904dfdaa080  2
PROCESS ffff8904dfdaa080
    SessionId: 0    Cid: 02e8    Peb: 8074164000    ParentCid: 0250
    DirBase: 3e590002    ObjectTable: ffffb00d0f4dab00    HandleCount:  42.
    Image: LsaIso.exe

        THREAD ffff8904dfdd9080  Cid 02e8.02f8  Teb: 0000008074165000
        Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Non-Alertable
            ffff8904dfdc5ca0 NotificationEvent

        THREAD ffff8904e12ac040  Cid 02e8.0b84  Teb: 0000008074167000
        Win32Thread: 0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffff8904dfdd7440 QueueObject

lkd> .thread /p ffff8904e12ac040
Implicit thread is now ffff8904`e12ac040
Implicit process is now ffff8904`dfdaa080
.cache forcedecodeuser done
lkd> k
  *** Stack trace for last set context  - .thread/.cxr resets it
 #  Child-SP          RetAddr           Call Site
00  ffffe009`1216c140  fffff801`27564e17  nt!KiSwapContext+0x76
01  ffffe009`1216c280  fffff801`27564989  nt!KiSwapThread+0x297
02  ffffe009`1216c340  fffff801`275681f9  nt!KiCommitThreadWait+0x549
03  ffffe009`1216c3e0  fffff801`27567369  nt!KeRemoveQueueEx+0xb59
04  ffffe009`1216c480  fffff801`27568e2a  nt!IoRemoveIoCompletion+0x99
05  ffffe009`1216c5b0  fffff801`2764d504  nt!NtWaitForWorkViaWorkerFactory+0x99a
06  ffffe009`1216c7e0  fffff801`276db75f  nt!VslpDispatchIumSyscall+0x34
07  ffffe009`1216c860  fffff801`27bab7e4  nt!VslpEnterIumSecureMode+0x12098b
08  ffffe009`1216c8d0  fffff801`276586cc  nt!PspUserThreadStartup+0x178704
09  ffffe009`1216c9c0  fffff801`27658640  nt!KiStartUserThread+0x1c
0a  ffffe009`1216cb00  00007fff`d06f7ab0  nt!KiStartUserThreadReturn
0b  00000080`7427fe18  00000000`00000000  ntdll!RtlUserThreadStart
```

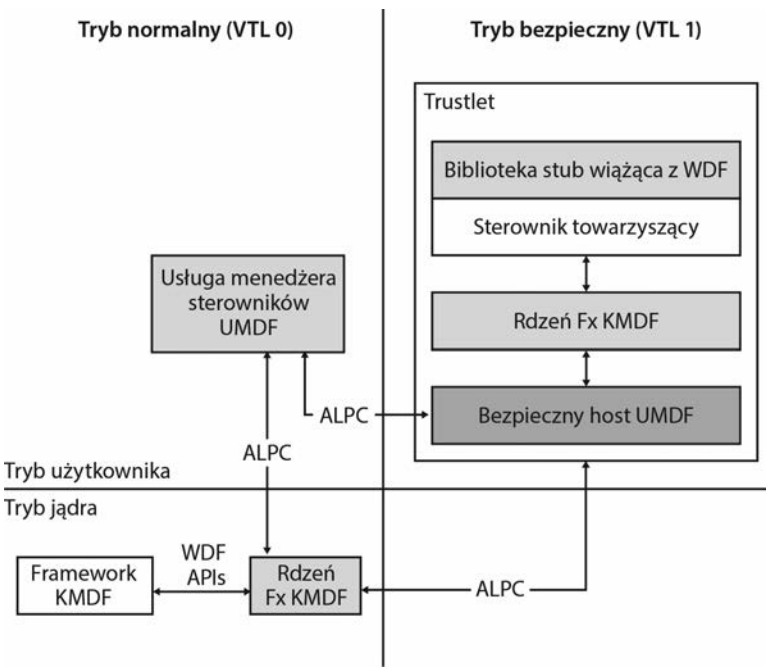
Stos wyraźnie pokazuje, że wykonanie rozpoczyna się w VTL 0 w procedurze *KiStartUserThread*. *PspUserThreadStartup* wywołał pętlę rozsyłania bezpiecznych połączeń, która nigdy się nie zakończyła i została przerwana przez operację `WAIT`. Nie ma możliwości, aby debuger jądra pokazał jakiekolwiek struktury danych bezpiecznego jądra lub prywatne dane trustletu.

Bezpieczne urządzenia

VBS zapewnia sterownikom możliwość uruchomienia części ich kodu w bezpiecznym środowisku. Samo bezpieczne jądro nie może być rozszerzone o obsługę sterowników jądra, jego powierzchnia narażona na atak stałaby się zbyt duża. Ponadto Microsoft nie pozwoliłby firmom zewnętrznym na wprowadzenie ewentualnych błędów do komponentu używanego głównie do celów bezpieczeństwa.

Framework sterowników trybu użytkownika (UMDF — ang. *User-Mode Driver Framework*) rozwiązuje ten problem, wprowadzając pojęcie *towarzyszy sterowników* (ang. *driver companions*), którzy mogą działać zarówno w trybie użytkownika VTL 0, jak i VTL 1. W tym przypadku przyjmują one nazwę *bezpiecznych towarzyszy* (ang. *secure companions*). Bezpieczny towarzysz pobiera podzbiór kodu sterownika, który musi działać w innym trybie (w tym przypadku IUM), i ładuje go jako rozszerzenie (lub towarzysza) głównego sterownika KMDF. Standardowe sterowniki WDM są jednak również obsługiwane. Główny sterownik, który nadal działa w trybie jądra VTL 0, wciąż zarządza stanem PnP i zasilania urządzenia, ale potrzebuje możliwości dotarcia do swojego towarzysza, aby wykonać zadania, które muszą być wykonane w IUM.

Chociaż wspomniany w rozdziale 3. framework bezpiecznych sterowników (SDF — ang. *Secure Driver Framework*) jest przestarzały, rysunek 9.39 przedstawia architekturę nowego modelu bezpiecznego towarzysza UMDF, który nadal jest zbudowany na tym samym frameworku rdzenia UMDF (*Wudfx02000.dll*) używanego w trybie użytkownika VTL 0. Ten ostatni wykorzystuje usługi świadczone przez hosta bezpiecznego towarzysza UMDF (*WUDFCompanionHost.exe*) do ładowania i zarządzania towarzyszem sterownika, który jest dystrybuowany za pośrednictwem biblioteki DLL. Host bezpiecznego towarzysza UMDF zarządza czasem życia bezpiecznego towarzysza i hermetyzuje wiele funkcji UMDF, które dotyczą środowiska IUM.



RYSUNEK 9.39. *Architektura bezpiecznego towarzysza sterownika WDF*

Bezpieczny towarzysz zwykle jest związany z głównym sterownikiem, który działa w jądrze VTL 0. Musi być odpowiednio podpisany (włączając EKU IUM do podpisu, jak dla każdego trustletu) i musi zadeklarować swoje możliwości w sekcji metadanych. Bezpieczny towarzysz ma pełną własność zarządzanego przez siebie urządzenia (to wyjaśnia, dlaczego urządzenie jest często nazywane *bezpiecznym urządzeniem*). Sterownik bezpiecznego urządzenia poprzez bezpiecznego towarzysza obsługuje następujące funkcje:

- **Bezpieczne DMA** — Sterownik może polecić urządzeniu wykonanie transferu DMA bezpośrednio w chronionej pamięci VTL 1, która nie jest dostępna dla VTL 0. Bezpieczny towarzysz może przetwarzać dane wysyłane lub odbierane przez interfejs DMA, a następnie może przekazać część danych do sterownika VTL 0 przez standardowy interfejs komunikacyjny KMDF (ALPC). Bezpieczne wywołania systemowe *IumGetDmaEnabler* i *IumDmaMapMemory*, prezentowane poprzez *Iumbase.dll*, pozwalają bezpiecznemu towarzyszowi mapować fizyczne zakresy pamięci DMA bezpośrednio w trybie użytkownika VTL 1.
- **We/wy mapowane pamięcią (MMIO — ang. *memory mapped IO*)** — Bezpieczny towarzysz może poprosić urządzenie o zmapowanie jego dostępnego zakresu MMIO w VTL 1 (tryb użytkownika). Następnie może uzyskać dostęp do rejestrów urządzenia mapowanego pamięciowo bezpośrednio w IUM. *MapSecureIo* i *ProtectSecureIo* API prezentują tę funkcję.
- **Bezpieczne sekcje** — Towarzysz może tworzyć (poprzez API *CreateSecureSection*) i mapować bezpieczne sekcje odzwierciedlające pamięć, która może być współdzielona pomiędzy trustletami i głównym sterownikiem działającym w VTL 0. Ponadto bezpieczny towarzysz może określić inny typ ochrony SLAT w przypadku, gdy dostęp do pamięci odbywa się za pośrednictwem bezpiecznego urządzenia (poprzez DMA lub MMIO).

Bezpieczny towarzysz nie może bezpośrednio odpowiadać na przerwania urządzenia, które muszą być mapowane i zarządzane przez powiązany sterownik trybu jądra w VTL 0. W ten sam sposób sterownik trybu jądra nadal musi działać jako interfejs wysokiego poziomu dla systemu i aplikacji trybu użytkownika poprzez zarządzanie wszystkimi otrzymanymi IOCTL-ami. Główny sterownik komunikuje się ze swoim bezpiecznym towarzyszem, wysyłając zadania WDF za pomocą obiektu Kolejka zadań UMDF, który wewnętrznie korzysta z udogodnień ALPC prezentowanych przez framework WDF.

Typowy sterownik KMDF rejestruje swojego towarzysza za pomocą dyrektyw INF. WDF automatycznie uruchamia towarzysza sterownika w kontekście jego wywołania *WdfDeviceCreate* — co w przypadku sterowników typu plug and play zwykle dzieje się w wywołaniu zwrotnym AddDevice — poprzez wysłanie wiadomości ALPC do usługi menedżera sterowników UMDF, która tworzy nowy trustlet *WUDFCompanionHost.exe* na skutek wywołania natywnego API *NtCreateUserProcess*. Następnie host bezpiecznego towarzysza UMDF ładuje bibliotekę DLL bezpiecznego towarzysza do swojej przestrzeni adresowej. Kolejna wiadomość ALPC jest wysyłana z menedżera sterowników UMDF do WUDFCompanionHost, w celu faktycznego uruchomienia bezpiecznego towarzysza. Procedura *DriverEntry* towarzysza wykonuje bezpieczną inicjalizację sterownika i tworzy obiekt WDFDRIVER poprzez klasyczne API *WdfDriverCreate*.

Następnie framework wywołuje procedurę wywołania zwrotnego AddDevice towarzysza w VTL 1, która zwykle tworzy urządzenie towarzysza poprzez nowe API *WdfDeviceCompanionCreate* UMDF. To ostatnie przekazuje wykonanie do bezpiecznego jądra (poprzez bezpieczne wywołanie systemowe *IumCreateSecureDevice*), które tworzy nowe bezpieczne urządzenie. Od tego momentu bezpieczny

towarzysz ma pełną własność zarządzanego przez siebie urządzenia. Zazwyczaj pierwszą rzeczą, jaką robi towarzysz po utworzeniu bezpiecznego urządzenia, jest utworzenie obiektu kolejki zadań (*WDFTASKQUEUE*) używanego do przetwarzania wszelkich przychodzących zadań dostarczanych przez jego powiązany sterownik VTL 0. Kontrola wykonania wraca do sterownika trybu jądra, który może teraz wysyłać nowe zadania do swojego bezpiecznego towarzysza.

Model ten jest również obsługiwany przez sterowniki WDM. Sterowniki WDM mogą używać trybu miniportu KMDF do interakcji ze specjalnym sterownikiem filtra, *WdmCompanionFilter.sys*, który jest dołączony na niższej pozycji stosu urządzenia. Filtr Wdm Companion umożliwia sterownikom WDM korzystanie z obiektu kolejki zadań do wysyłania zadań do bezpiecznego towarzysza.

Enklawy oparte na VBS

W rozdziale 5. części I omówiliśmy rozszerzenie Software Guard Extensions (SGX), technologię sprzętową umożliwiającą tworzenie chronionych enklaw pamięci, czyli bezpiecznych stref w przestrzeni adresowej procesu, w których kod i dane są chronione (szyfrowane) przez sprzęt przed kodem działającym poza enklawą. Technologia, która została po raz pierwszy wprowadzona w procesorach Intel Core szóstej generacji (Skylake), cierpiała na pewne problemy, które uniemożliwiły jej szerokie przyjęcie. (Ponadto AMD wypuściło inną technologię, znaną jako bezpieczna szyfrowana wirtualizacja (ang. *Secure Encrypted Virtualization*), która nie jest kompatybilna z SGX).

Aby przezwyciężyć te problemy, firma Microsoft wypuściła enklawy oparte na VBS, czyli bezpieczne enklawy, których gwarancje izolacji są zapewnione przy użyciu infrastruktury wirtualnego trybu bezpiecznego (VSM — ang. *Virtual Secure Mode*). Kod i dane wewnątrz enklawy opartej na VBS są widoczne tylko dla samej enklawy (oraz bezpiecznego jądra VSM) i są niedostępne dla jądra NT, procesów VTL 0 i bezpiecznych trustletów działających w systemie.

Bezpieczna enklawa oparta na VBS jest tworzona poprzez utworzenie pojedynczego wirtualnego zakresu adresów w ramach normalnego procesu. Kod i dane są następnie ładowane do enklawy, po czym następuje pierwsze wejście do enklawy poprzez przekazanie kontroli do jej punktu wejścia przez bezpieczne jądro. Bezpieczne jądro najpierw weryfikuje, że wszystkie kody i dane są autentyczne i są upoważnione do działania wewnątrz enklawy poprzez zastosowanie weryfikacji podpisu obrazu enklawy. Jeśli weryfikacja podpisu przebiegnie pomyślnie, wtedy kontrola wykonania jest przekazywana do punktu wejścia enklawy, który ma dostęp do całego kodu i danych enklawy. Domyślnie system obsługuje tylko wykonanie enklaw, które są prawidłowo podpisane. Wyklucza to możliwość wykonania niepodpisanego złośliwego oprogramowania w systemie poza zasięgiem wzroku oprogramowania antywirusowego, które nie jest w stanie sprawdzić zawartości żadnej enklawy.

Podczas wykonywania kodu kontrola może być przekazywana w obie strony między enklawą a jej procesem wewnętrznym. Kod wykonywany wewnątrz enklawy ma dostęp do wszystkich danych w wirtualnym zakresie adresów enklawy. Ponadto ma dostęp do odczytu i zapisu w przestrzeni adresowej zawierającej niezabezpieczony proces. Cała pamięć w zakresie adresów wirtualnych enklawy będzie niedostępna dla procesu wewnętrznego. Jeśli wiele enklaw istnieje w ramach pojedynczego procesu hosta, każda enklawa będzie mogła uzyskać dostęp tylko do własnej pamięci i pamięci, która jest dostępna dla procesu hosta.

Tak jak w przypadku enklaw sprzętowych, kiedy kod działa w enklawie, może uzyskać raport o zamkniętej enklawie, który może być wykorzystany przez podmiot zewnętrzny do sprawdzenia, czy kod działa z gwarancjami izolacji enklawy VBS, i który może być dalej wykorzystany do sprawdzenia konkretnej wersji działającego kodu. Raport ten zawiera informacje o systemie hosta, samej enklawie i wszystkich DLL-ach, które mogły zostać załadowane do enklawy, jak również informacje wskazujące, czy enklawa wykonuje kod z włączonymi możliwościami debugowania.

Enklawa oparta na VBS jest dystrybuowana jako DLL i ma pewne specyficzne cechy:

- Jest podpisana podpisem *authenticode*, a certyfikat liścia zawiera ważne EKU, które pozwala na uruchomienie obrazu jako enklawy. Głównym organem autoryzującym, który wydał certyfikat cyfrowy, powinien być Microsoft lub organ podpisujący strony trzeciej, objęty manifestem certyfikatu, który jest kontrasygnowany przez Microsoft. Oznacza to, że firmy trzecie mogą podpisywać i uruchamiać swoje własne enklawy. Prawidłowe EKU podpisu cyfrowego to IUM EKU (1.3.6.1.4.1.311.10.3.37) dla wewnętrznych enklaw podpisanych przez Windows lub Enclave EKU (1.3.6.1.4.1.311.10.3.42) dla wszystkich enklaw stron trzecich.
- Zawiera sekcję konfiguracji enklawy (odzwierciedlaną przez strukturę danych *IMAGE_ENCLAVE_CONFIG*), która opisuje informacje o enklawie i która jest powiązana z katalogiem danych konfiguracji jej załadowanego obrazu.
- Zawiera prawidłowe oprzyrządowanie Strażnika kontroli przepływu (CFG — ang. *Control Flow Guard*).

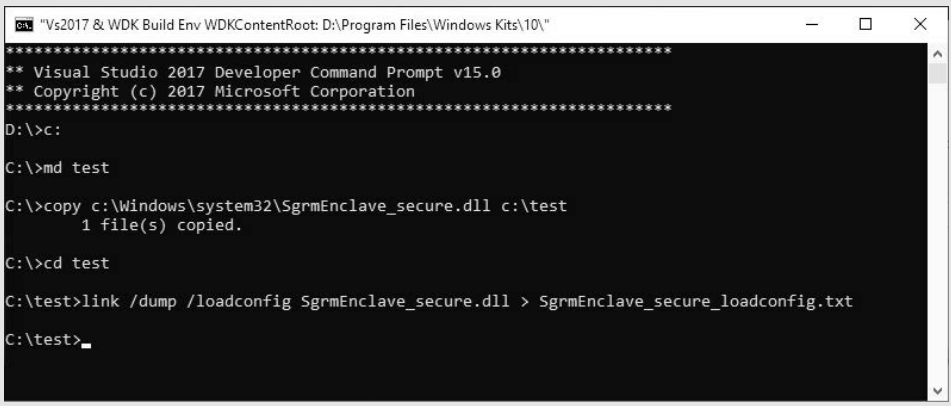
Sekcja konfiguracji enklawy jest istotna, ponieważ zawiera ważne informacje potrzebne do prawidłowego uruchomienia i zapieczętowania enklawy: unikalny identyfikator rodziny i identyfikator obrazu, które są określone przez autora enklawy i identyfikują binaria enklawy, numer bezpiecznej wersji i informacje o polityce enklawy (jak oczekiwany rozmiar wirtualny, maksymalna liczba wątków, które mogą być uruchomione, i zdolność do debugowania enklawy). Ponadto sekcja konfiguracyjna enklawy zawiera listę obrazów, które mogą być importowane przez enklawę, wraz z informacjami o ich tożsamości. Zaimportowany moduł enklawy może być identyfikowany przez kombinację ID rodziny i ID obrazu lub przez kombinację wygenerowanego unikalnego ID, które jest obliczane, począwszy od hasha binarnego, oraz ID autora, które pochodzi z certyfikatu użytego do podpisania enklawy. (Ta wartość wyraża tożsamość tego, kto skonstruował enklawę). Deskryptor importowanego modułu musi także zawierać minimalny numer bezpiecznej wersji.

Bezpieczne jądro oferuje pewne podstawowe usługi systemowe dla enklaw poprzez DLL wykonawcze enklawy VBS, *Vertdll.dll*, które jest mapowane w przestrzeni adresowej enklawy. Usługi te obejmują: ograniczony podzbiór standardowej biblioteki wykonawczej C, możliwość przydzielenia lub zwolnienia bezpiecznej pamięci w zakresie adresów enklawy, usługi synchronizacji, wsparcie dla obsługi wyjątków strukturalnych, podstawowe funkcje kryptograficzne oraz możliwość zapieczętowania danych.

Eksperyment: dumping (zrzut) konfiguracji enklawy

W tym eksperymencie wykorzystujemy linker Microsoft Incremental (*link.exe*) dołączony do Windows SDK i WDK do zrzucania danych konfiguracyjnych enklawy programowej. Oba pakiety można pobrać z sieci. Można też skorzystać z EWDK, który zawiera wszystkie niezbędne narzędzia i nie wymaga instalacji. Jest on dostępny pod adresem *https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk*.

Otwieramy okno wiersza poleceń dla deweloperów programu Visual Studio korzystając z okna wyszukiwania Cortany lub wykonując plik skryptu *LaunchBuildEnv.cmd* zawarty w obrazie ISO pakietu EWDK. Przeanalizujemy dane konfiguracyjne enklawy System Guard Routine Attestation — która jest pokazana na rysunku 9.40 i zostanie opisana w dalszej części tego rozdziału — za pomocą polecenia `link /dump /loadconfig`:



Dane wyjściowe polecenia są duże. Dlatego w przykładzie pokazanym na poprzednim rysunku przekierowaliśmy je do pliku *SgrmEnclave_secure_loadconfig.txt*. Jeśli otworzymy nowy plik wyjściowy, zobaczymy, że obraz binarny zawiera tablicę CFG i poprawny wskaźnik konfiguracji enklawy, który kieruje do następujących danych:

```
Enclave Configuration

            00000050 size
            0000004C minimum required config size
            00000000 policy flags
            00000003 number of enclave import descriptors
            0004FA04 RVA to enclave import descriptors
            00000050 size of an enclave import descriptor
            00000001 image version
            00000001 security version
     0000000010000000 enclave size
            00000008 number of threads
            00000001 enclave flags

            family ID : B1 35 7C 2B 69 9F 47 F9 BB C9 4F 44 F2 54 DB 9D
             image ID : 24 56 46 36 CD 4A D8 86 A2 F4 EC 25 A9 72 02

    ucrtbase_enclave.dll
```

```
            0 minimum security version
            0 reserved

              match type : image ID
               family ID : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                image ID : F0 3C CD A7 E8 7B 46 EB AA E7 1F 13 D5 CD DE 5D
      unique/author ID : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

bcrypt.dll

            0 minimum security version
            0 reserved

              match type : image ID
               family ID : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                image ID : 20 27 BD 68 75 59 49 B7 BE 06 34 50 E2 16 D7 ED
      unique/author ID : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Sekcja konfiguracyjna zawiera dane enklawy obrazu binarnego (jak identyfikator rodziny, identyfikator obrazu i numer wersji zabezpieczeń) oraz tablicę deskryptorów importu, która komunikuje bezpiecznemu jądru, od której biblioteki licząc, binarny obraz głównej enklawy może bezpiecznie polegać na danej bibliotece. Możesz powtórzyć eksperyment z biblioteką *Vertdll.dll* i ze wszystkimi binariami importowanymi z enklawy atestacji procedur strażnika systemu (ang. *System Guard Routine Attestation*).

Cykl życia enklawy

W rozdziale 5. części I omówiliśmy cykl życia enklawy sprzętowej (opartej na rozszerzeniu „strażnik oprogramowania" (SGX — ang. *Software Guard Extension*)). Cykl życia enklawy zbudowanej na bazie mechanizmu VBS, czyli bezpieczeństwo oparte na wirtualizacji (ang. *Virtualization-based security*) jest podobny; Microsoft rozszerzył dostępne już interfejsy API enklawy, aby obsługiwać nowy typ enklaw opartych na VBS.

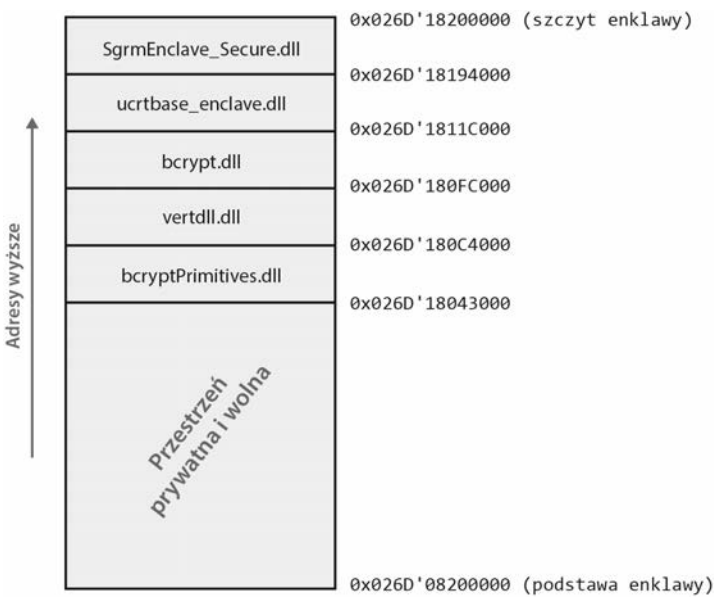
Krok 1: tworzenie. Aplikacja tworzy enklawę opartą na VBS poprzez określenie flagi `ENCLAVE_TYPE_VBS` do *API CreateEnclave*. Wywołujący powinien określić ID właściciela, które identyfikuje właściciela enklawy. Kod tworzący enklawę, w taki sam sposób jak w przypadku enklaw sprzętowych, kończy się wywołaniem *NtCreateEnclave* w jądrze. Ten ostatni sprawdza parametry, kopiuje przekazane struktury i dołącza do procesu docelowego na wypadek, gdyby enklawa miała być utworzona w innym procesie niż proces wywołujący. Funkcja *MiCreateEnclave* przydziela VAD opisujący zakres pamięci wirtualnej enklawy i wybiera bazowy adres wirtualny, jeśli nie został on określony przez wywołującego. Jądro przydziela strukturę danych enklawy VBS menedżera pamięci oraz tablicę haseł enklawy dla każdego procesu, używaną do szybkiego wyszukiwania enklawy rozpoczynającej się od jej numeru. Jeśli jest to pierwsza enklawa utworzona dla procesu, system tworzy również pusty bezpieczny proces (który działa jako kontener dla enklaw) w VTL 1 za pomocą bezpiecznego wywołania *CREATE_PROCESS* (więcej szczegółów we wcześniejszym punkcie „Tworzenie enklaw").

Bezpieczny handler wywołania *CREATE_ENCLAVE* w VTL 1 wykonuje faktyczną pracę przy tworzeniu enklawy: przydziela strukturę danych klucza bezpiecznej enklawy (*SKMI_ENCLAVE*), ustawia referencję do bezpiecznego procesu kontenera (który właśnie został utworzony przez

jądro NT) i tworzy bezpieczny VAD opisujący całą wirtualną przestrzeń adresową enklawy (bezpieczny VAD zawiera podobne informacje jak jego odpowiednik w VTL 0). Ten VAD jest wstawiany do drzewa VAD procesu wewnętrznego (a nie do samej enklawy). Pusta wirtualna przestrzeń adresowa dla enklawy jest tworzona w taki sam sposób jak dla procesu wewnętrznego: root tablicy stron jest wypełniany tylko wpisami systemowymi.

Krok 2: ładowanie modułów do enklawy. Inaczej niż w przypadku enklaw sprzętowych, proces nadrzędny może ładować do enklawy tylko moduły, ale nie arbitralne dane. Spowoduje to, że każda strona obrazu zostanie skopiowana do przestrzeni adresowej w VTL 1. Każda strona obrazu w enklawie VTL 1 będzie prywatną kopią. Przynajmniej jeden moduł (pełniący rolę głównego obrazu enklawy) musi być załadowany do enklawy; w przeciwnym razie enklawa nie może być zainicjalizowana. Po utworzeniu enklawy VBS aplikacja wywołuje API *LoadEnclaveImage*, podając adres bazowy enklawy i nazwę modułu, który musi być załadowany do enklawy. Kod programu Windows Loader (w *Ntdll.dll*) wyszukuje podaną nazwę DLL, otwiera i sprawdza poprawność jej pliku binarnego i tworzy obiekt sekcji, który jest mapowany z dostępem tylko do odczytu bezpośrednio w procesie wywołującym.

Po odwzorowaniu sekcji program ładujący parsuje tablicę adresów importu obrazu w celu utworzenia listy zależnych modułów (importowanych, załadowanych z opóźnieniem i przekazanych dalej). Dla każdego znalezionego modułu program ładujący sprawdza, czy w enklawie jest wystarczająco dużo miejsca na jego mapowanie i oblicza prawidłowy adres bazowy obrazu. Jak pokazano na rysunku 9.40, który przedstawia enklawę atestacji wykonawczej strażnika systemu (ang. *System Guard Runtime Attestation*), moduły w enklawie są mapowane przy użyciu strategii góra-dół (ang. *top-down*). Oznacza to, że główny obraz mapowany jest na najwyższym możliwym adresie wirtualnym, a wszystkie zależne mapowane są w niższych adresach jeden obok drugiego. Na tym etapie, dla każdego modułu Windows Loader wywołuje API jądra *NtLoadEnclaveData*.



RYSUNEK 9.40. *Bezpieczna enklawa atestacji wykonawczej strażnika systemu (zauważ puste miejsce u podstawy enklawy)*

W celu załadowania określonego obrazu do enklawy VBS jądro uruchamia złożony proces, który pozwala na skopiowanie stron współdzielonych jego obiektu sekcji do stron prywatnych enklawy w VTL 1. Funkcja *MiMapImageForEnclaveUse* pobiera obszar kontrolny obiektu sekcji i sprawdza go poprzez mechanizm bezpiecznej integralności kodu jądra (SKCI — ang. *Secure Kernel Code Integrity*). Jeśli walidacja się nie powiedzie, proces zostaje przerwany, a do wywołującego zwracany jest błąd. (Wszystkie moduły enklawy powinny być poprawnie podpisane, jak omówiono to wcześniej). W przeciwnym przypadku system dołącza do bezpiecznego procesu systemowego i mapuje obiekt sekcji obrazu w swojej przestrzeni adresowej w VTL 0. Strony współdzielone modułu w tym czasie mogą być ważne lub nieważne; więcej szczegółów w rozdziale 5. części I. Następnie deklaruje (ang. *commit*) wirtualną przestrzeń adresową modułu w procesie wewnętrznym. Tworzy to prywatne struktury danych stronicowania VTL 0 dla PTE z zerowym zapotrzebowaniem, które będą później wypełnione przez bezpieczne jądro, gdy obraz zostanie załadowany w VTL 1.

Bezpieczny handler wywołania *LOAD_ENCLAVE_MODULE* w VTL 1 uzyskuje „bezpieczny obraz" (ang. *SECURE_IMAGE*) nowego modułu (utworzonego przez SKCI) i sprawdza, czy obraz nadaje się do użycia w enklawie opartej na VBS (przez sprawdzenie charakterystyki podpisu cyfrowego). Następnie dołącza do bezpiecznego procesu systemowego w VTL 1 i mapuje bezpieczny obraz pod tym samym wirtualnym adresem, który wcześniej został zmapowany przez jądro NT. Pozwala to na współdzielenie prototypowych PTE z VTL 0. Bezpieczne jądro tworzy następnie bezpieczny VAD opisujący moduł i wstawia go do przestrzeni adresowej VTL 1 enklawy. Na koniec cyklicznie przechodzi pomiędzy prototypowymi PTE sekcji każdego modułu. Dla każdego nieobecnego prototypu PTE dołącza do bezpiecznego procesu systemowego i za pomocą normalnego wywołania *GET_PHYSICAL_PAGE* wywołuje obsługę błędu strony NT (*MmAccessFault*), która wprowadza do pamięci stronę współdzieloną. Bezpieczne jądro wykonuje podobny proces dla stron enklawy prywatnej, które zostały wcześniej zadeklarowane przez jądro NT w VTL 0 przez żądania zerowe PTE. Obsługa błędów stron NT w tym przypadku przydziela wyzerowane strony. Bezpieczne jądro kopiuje zawartość każdej współdzielonej strony fizycznej do każdej nowej strony prywatnej i w razie potrzeby stosuje potrzebne relokacje prywatne.

Ładowanie modułu w enklawie opartej na VBS jest zakończone. Bezpieczne jądro stosuje ochronę SLAT do prywatnych stron enklawy (jądro NT nie ma dostępu do kodu i danych obrazu w enklawie), usuwa mapowanie sekcji współdzielonej z bezpiecznego procesu systemowego i przekazuje wykonanie na rzecz jądra NT. Program ładujący (loader) może teraz przejść do następnego modułu.

Krok 3: inicjalizacja enklawy. Po załadowaniu wszystkich modułów do enklawy aplikacja inicjalizuje enklawę za pomocą *InitializeEnclave* API i określa maksymalną liczbę wątków obsługiwanych przez enklawę (które będą związane z wątkami zdolnymi do wykonywania wywołań enklawy w procesie wewnętrznym). Program obsługi bezpiecznego wywołania *INITIALIZE_ENCLAVE* bezpiecznego jądra sprawdza, czy zasady określone podczas tworzenia enklawy są zgodne z zasadami wyrażonymi w informacjach konfiguracyjnych obrazu głównego, sprawdza, czy biblioteka platformy enklawy jest załadowana (*Vertdll.dll*), oblicza końcowy 256-bitowy hash enklawy (używany do generowania raportu zapieczętowania enklawy) i tworzy wszystkie bezpieczne wątki enklawy. Gdy kontrola wykonania jest zwracana do kodu programu Windows Loader w VTL 0, system wykonuje pierwsze wywołanie enklawy, które wykonuje kod inicjalizacyjny plików DLL platformy.

Krok 4: wywołania enklawy (przychodzące i wychodzące). Po prawidłowym zainicjalizowaniu enklawy aplikacja może wykonać dowolną liczbę wywołań do enklawy. Wszystkie wywoływane funkcje w enklawie muszą być wyeksportowane. Aplikacja może wywołać standardowe API *GetProcAddress*, aby uzyskać adres funkcji enklawy, a następnie użyć procedury *CallEnclave* do przekazania kontroli wykonania do bezpiecznej enklawy. W tym scenariuszu, który opisuje *wywołanie przychodzące*, procedura jądra *NtCallEnclave* wykonuje algorytm wyboru wątku, który wiąże wywołujący wątek VTL 0 z wątkiem enklawy, zgodnie z następującymi zasadami:

- Jeśli normalny wątek nie został wcześniej wywołany przez enklawę (enklawy obsługują wywołania zagnieżdżone), to do wykonania wybierany jest dowolny bezczynny wątek enklawy. W przypadku gdy nie są dostępne żadne bezczynne wątki enklawy, wywołanie blokuje się do czasu, gdy wątek enklawy stanie się dostępny (jeśli został określony przez wywołującego; w przeciwnym razie wywołanie po prostu się nie powiedzie).
- W przypadku gdy normalny wątek został wcześniej wywołany przez enklawę, to wywołanie do enklawy odbywa się na tym samym wątku enklawy, który wydał poprzednie wywołanie do hosta.

Lista deskryptorów wątku enklawy jest utrzymywana zarówno przez NT, jak i bezpieczne jądro. Gdy normalny wątek jest związany z wątkiem enklawy, wątek enklawy jest umieszczany na innej liście, która jest nazywana listą *wątków związanych*. Wątki enklaw śledzone przez tę ostatnią listę są aktualnie uruchomione i nie są już dostępne.

Po pomyślnym wykonaniu algorytmu wyboru wątków jądro NT emituje bezpieczne wywołanie *CALLENCLAVE*. Bezpieczne jądro tworzy nową ramkę stosu dla enklawy i powraca do trybu użytkownika. Pierwszą funkcją trybu użytkownika wykonywaną w kontekście enklawy jest *RtlEnclaveCallDispatcher*. Ta ostatnia, w przypadku gdy wywołanie enklawy było pierwszym w historii, przenosi wykonanie do procedury inicjalizacyjnej biblioteki uruchomieniowej DLL enklawy VSM (*Vertdll.dll*), która to biblioteka inicjalizuje CRT, Loader i wszystkie usługi dostarczane enklawie; na koniec funkcja ta wywołuje funkcję *DllMain* głównego modułu enklawy i wszystkich jej zależnych obrazów (poprzez określenie przyczyny *DLL_PROCESS_ATTACH*).

W normalnych sytuacjach, gdy biblioteka DLL platformy enklawy została już zainicjalizowana, dyspozytor enklawy wywołuje *DllMain* każdego modułu podając powód *DLL_THREAD_ATTACH*, następnie sprawdza, czy podany adres funkcji docelowej enklawy jest poprawny, a jeśli tak, to ostatecznie wywołuje funkcję docelową. Gdy procedura enklawy docelowej zakończy swoje działanie, powraca do VTL 0 przez wywołanie z powrotem do procesu wewnętrznego. W tym celu nadal opiera się na bibliotece DLL platformy enklawy, która to biblioteka ponownie wywołuje procedurę jądra *NtCallEnclave*. Chociaż ta ostatnia jest zaimplementowana nieco inaczej w bezpiecznym jądrze, przyjmuje podobną strategię powrotu do VTL 0. Sama enklawa może emitować wywołania enklawy w celu wykonania jakiejś funkcji w kontekście niezabezpieczonego procesu wewnętrznego. W tym scenariuszu (który opisuje wywołanie wychodzące), kod enklawy używa procedury *CallEnclave* i określa adres wyeksportowanej funkcji w głównym module procesu wewnętrznego.

Krok 5: zakończenie i zniszczenie. Gdy zakończenie całej enklawy jest żądane poprzez API *TerminateEnclave*, wszystkie wątki wykonywane wewnątrz enklawy będą zmuszone do powrotu do VTL 0. Po zażądaniu zakończenia enklawy wszystkie dalsze wywołania do enklawy zakończą się niepowodzeniem. Gdy wątki kończą pracę, ich stan wątku VTL 1 (w tym stosy wątków) jest niszczony. Gdy wszystkie wątki przestaną wykonywać instrukcje, enklawa może zostać zniszczona. Gdy enklawa jest niszczona, wszystkie pozostałe stany VTL 1 związane z enklawą są również niszczone (w tym cała przestrzeń adresowa enklawy), a wszystkie strony są zwalniane

w VTL 0. Wreszcie VAD enklawy jest usuwany i cała zadeklarowane pamięć enklawy jest uwalniana. Destrukcja jest uruchamiana, gdy proces wewnętrzny wywołuje *VirtualFree* z podstawą zakresu adresowego enklawy. Destrukcja nie jest możliwa, chyba że enklawa została zakończona lub nigdy nie została zainicjalizowana.

Uwaga. Jak omówiliśmy to wcześniej, wszystkie strony pamięci, które są mapowane do przestrzeni adresowej enklawy, są prywatne. Niesie to ze sobą wiele konsekwencji. W przestrzeni adresowej enklawy nie są jednak mapowane żadne strony pamięci, które należą do procesu wewnętrznego VTL 0 (a także nie są obecne żadne VAD opisujące przydzielenie procesu wewnętrznego). Jak więc enklawa może uzyskać dostęp do wszystkich stron pamięci procesu wewnętrznego?

Odpowiedź znajduje się w handlerze błędów stron bezpiecznego jądra (*SkmmAccessFault*). W swoim kodzie handler błędów sprawdza, czy proces powodujący błąd jest enklawą. Jeśli tak, to handler błędów sprawdza, czy błąd wystąpił, ponieważ enklawa próbowała wykonać jakiś kod poza swoim regionem. W takim przypadku handler wywołuje błąd naruszenia dostępu. Jeśli błąd jest spowodowany dostępem do odczytu lub zapisu poza przestrzenią adresową enklawy, to handler błędów bezpiecznej strony emituje normalną usługę *GET_PHYSICAL_PAGE*, co powoduje wywołanie handlera błędu dostępu VTL 0. Handler VTL 0 sprawdza drzewo VAD procesu wewnętrznego, uzyskuje PFN strony z jego PTE — w razie potrzeby wprowadzając go do pamięci — i zwraca go do VTL 1. Na tym etapie bezpieczne jądro może stworzyć niezbędne struktury stronicowania, aby zmapować stronę fizyczną do tego samego adresu wirtualnego (którego dostępność jest gwarantowana dzięki właściwości samej enklawy) i wznowić wykonanie instrukcji. Strona jest teraz ważna w kontekście bezpiecznej enklawy.

Pieczętowanie i atestacja (poświadczanie)

Enklawy oparte na VBS, podobnie jak enklawy sprzętowe, obsługują zarówno pieczętowanie, jak i atestację (poświadczanie) danych. Termin *pieczętowanie* odnosi się do szyfrowania dowolnych danych przy użyciu jednego lub większej liczby kluczy szyfrujących, które nie są widoczne dla kodu enklawy, ale są zarządzane przez bezpieczne jądro i powiązane z maszyną oraz tożsamością enklawy. Enklawy nigdy nie będą miały dostępu do tych kluczy; zamiast tego bezpieczne jądro oferuje usługi pieczętowania i odpieczętowywania dowolnej zawartości (poprzez interfejsy API *EnclaveSealData* i *EnclaveUnsealData*) przy użyciu odpowiedniego klucza wskazanego przez enklawę. W czasie gdy dane są zapieczętowane, dostarczany jest zestaw parametrów, który kontroluje, które enklawy mają prawo do odpieczętowania danych. Obsługiwane są następujące zasady:

- **Numer wersji bezpieczeństwa (SVN) bezpiecznego jądra i obrazu głównego** — żadna enklawa nie może odpieczętować danych, które zostały zapieczętowane przez późniejszą wersję enklawy lub bezpiecznego jądra.
- **Dokładny kod** — dane mogą być odpieczętowane tylko przez enklawę odwzorowującą te same moduły enklawy, która je zapieczętowała. Bezpieczne jądro weryfikuje hash unikalnego ID każdego obrazu zmapowanego w enklawie, aby umożliwić prawidłowe odpieczętowanie.
- **Ten sam obraz, rodzina lub autor** — dane mogą być odpieczętowane tylko przez enklawę, która ma ten sam ID autora, ID rodziny i/lub ID obrazu.
- **Zasada uruchomieniowa** — dane mogą być odpieczętowane tylko wtedy, gdy enklawa odpieczętowująca ma taką samą zasadę debugowania jak enklawa oryginalna (debugowalna versus niedebugowalna).

Każda enklawa może zaświadczyć dowolnej stronie trzeciej, że działa jako enklawa VBS ze wszystkimi zabezpieczeniami oferowanymi przez architekturę enklawy VBS. Raport atestacyjny enklawy dostarcza dowodu na to, że określona enklawa działa pod kontrolą bezpiecznego jądra. Raport atestacyjny zawiera tożsamość całego kodu załadowanego do enklawy, jak również zasady kontrolujące sposób wykonywania enklawy.

Opisywanie wewnętrznych szczegółów operacji pieczętowania i atestacji wykracza poza zakres tej książki. Enklawa może wygenerować raport atestacyjny poprzez interfejs API *EnclaveGetAttestationReport*. Bufor pamięci zwrócony przez API może być przekazany do innej enklawy, będącej w stanie „zaświadczyć" o integralności środowiska, w którym działała oryginalna enklawa, weryfikując raport atestacyjny za pomocą funkcji *EnclaveVerifyAttestationReport*.

Atestacja uruchomieniowa strażnika systemu

Atestacja uruchomieniowa strażnika systemu (SGRA — ang. *System Guard runtime attestation*) jest komponentem integralności systemu operacyjnego, który wykorzystuje wyżej wymienione enklawy VBS — wraz ze zdalnym komponentem usługi atestacyjnej — aby zapewnić silne gwarancje wokół swojego środowiska wykonawczego. To środowisko jest używane do oceniania wrażliwych właściwości systemu w czasie pracy i pozwala stronie ufającej obserwować naruszenia obietnic bezpieczeństwa, które zapewnia system. Pierwsza implementacja tej nowej technologii została wprowadzona w aktualizacji systemu Windows z 10 kwietnia 2018 (RS4).

Atestacja SGRA pozwala aplikacji wyświetlić oświadczenie o postawie bezpieczeństwa urządzenia. Oświadczenie to składa się z trzech części:

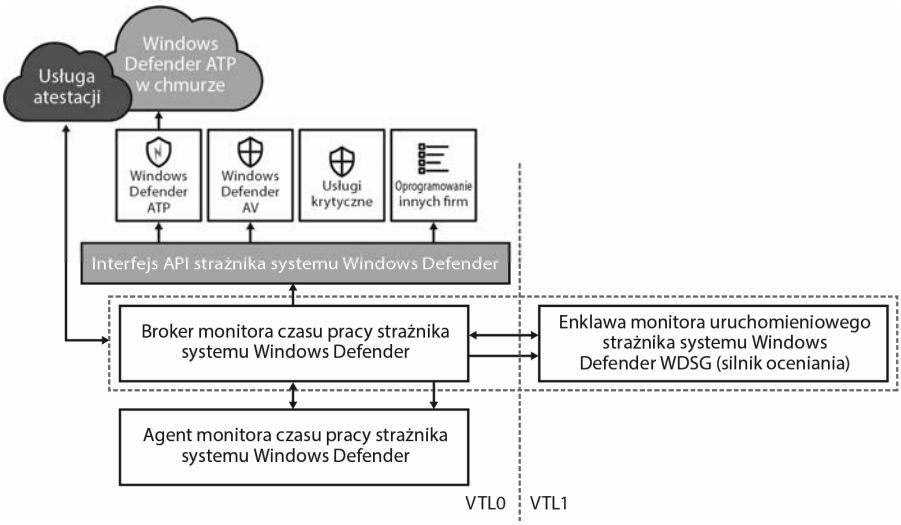
- raportu sesji, który zawiera poziom bezpieczeństwa opisujący atestowane właściwości rozruchowe (*boot-time*) urządzenia;
- raportu uruchomieniowego, który opisuje stan uruchomieniowy urządzenia;
- podpisanego certyfikatu sesji, który może być użyty do weryfikacji raportów.

Usługa atestacji SGRA, *SgrmBroker.exe*, hostuje komponent (*SgrmEnclave_secure.dll*), który działa w VTL 1 jako enklawa VBS, która stale ocenia system pod kątem naruszeń zabezpieczeń w trybie uruchomieniowym. Te oceny są ujawniane w raporcie uruchomieniowym, który może być weryfikowany po stronie back-endu przez część ufającą. Ponieważ oceny działają w oddzielnej domenie zaufania, bezpośredni atak na zawartość raportu uruchomieniowego staje się trudny.

Wewnętrzne mechanizmy atestacji SGRA

Rysunek 9.41 przedstawia wysokopoziomowy przegląd architektury atestacji uruchomieniowej strażnika systemu (SGRA) aplikacji Windows Defender, która składa się z następujących komponentów po stronie klienta:

- silnik oceniania VTL 1: *SgrmEnclave_secure.dll*;
- agent trybu jądra VTL 0: *SgrmAgent.sys*;
- proces zaufanej bazy komputera Windows WinTCB (ang. *Windows Trusted Computer Base*) chronionego brokera VTL 0 hostujący silnik oceniania: *SgrmBroker.exe*;
- proces LPAC VTL 0 używany przez proces brokera WinTCBPP do interakcji ze stosem sieciowym: *SgrmLpac.exe*.



RYSUNEK 9.41. *Architektura atestacji uruchomieniowej komponentu Windows Defender System Guard (strażnik systemu aplikacji Windows Defender)*

Aby móc szybko reagować na zagrożenia, atestacja SGRA zawiera dynamiczny silnik skryptowy (Lua), tworzący rdzeń mechanizmu oceniania, który jest wykonywany w enklawie VTL 1 — jest to podejście pozwalające na częste aktualizacje logiki oceniania.

Ze względu na izolację zapewnianą przez enklawę VBS wątki wykonywane w VTL 1 są ograniczone pod względem możliwości dostępu do interfejsów API jądra NT VTL 0. Dlatego aby komponent uruchomieniowy SGRA mógł wykonać sensowną pracę, konieczny jest sposób obejścia ograniczonej powierzchni API enklawy VBS.

W celu wystawienia obiektów VTL 0 na logikę działającą w VTL 1 zaimplementowano podejście oparte na agencie; obiekty te są określane jako asystenci i są obsługiwane przez komponent trybu użytkownika SgrmBroker lub przez sterownik agenta działający w trybie jądra VTL 0 (*SgrmAgent.sys*). Logika VTL 1 działająca w enklawie może wywołać te komponenty VTL 0 w celu zażądania asysty, która zapewnia szereg udogodnień, w tym prymitywy synchronizacji jądra NT, możliwości mapowania stron i tak dalej.

Jako przykład działania tego mechanizmu — atestacja SGRA jest w stanie pozwolić silnikowi potwierdzeń VTL 1 na bezpośrednie czytanie stron fizycznych należących do VTL 0. Enklawa żąda mapowania dowolnej strony za pomocą asysty. Strona zostanie następnie zablokowana i zmapowana do przestrzeni adresowej SgrmBroker VTL 0 (czyniąc ją rezydentną). Ponieważ enklawy VBS mają bezpośredni dostęp do przestrzeni adresowej procesu hosta, bezpieczna logika może czytać bezpośrednio z mapowanych adresów wirtualnych. Te odczyty muszą być synchronizowane z samym jądrem VTL 0. Do wykonania synchronizacji wykorzystywany jest również rezydentny agent brokera VTL 0 (sterownik *SgrmAgent.sys*).

Logika oceniania

Jak wspomniano wcześniej, atestacja SGRA ocenia właściwości bezpieczeństwa systemu podczas fazy uruchomieniowej (*runtime*). Oceny te są wykonywane w ramach silnika oceniania hostowanego w enklawie opartej na VBS. Podpisany kod bajtowy Lua opisujący logikę oceniania jest dostarczany do silnika oceniania podczas uruchamiania.

Oceny są uruchamiane okresowo. Kiedy odkryte zostanie naruszenie zapewnionej właściwości (czyli kiedy ocena się „nie powiedzie"), niepowodzenie jest rejestrowane i przechowywane w enklawie. To niepowodzenie będzie zaprezentowane (ujawnione) stronie ufającej w raporcie uruchomieniowym (*runtime*), który jest generowany i podpisywany (z certyfikatem sesji) w obrębie enklawy.

Przykładem możliwości oceniania dostarczanych przez atestację SGRA są oceny dotyczące różnych atrybutów obiektu procesu wykonawczego — na przykład okresowe wyliczanie działających procesów i ocena stanu bitów ochrony procesu, które rządzą zasadami chronionych procesów.

Przepływ silnika oceniania wykonującego to sprawdzenie może być w przybliżeniu przedstawiony za pomocą następujących kroków:

1. Silnik oceniania działający w VTL 1 wywołuje swój proces hosta VTL 0 (SgrmBroker), by zażądać od jądra odwołania do obiektu procesu wykonawczego.
2. Proces brokera przekazuje to żądanie do agenta trybu jądra (SgrmAgent), który obsługuje żądanie przez uzyskanie odniesienia do żądanego obiektu procesu wykonawczego.
3. Agent powiadamia brokera, że żądanie zostało obsłużone i przekazuje wszelkie niezbędne metadane w dół, do brokera.
4. Broker przekazuje tę odpowiedź do logiki oceniania VTL 1 wysyłającej żądania.
5. Logika może wtedy wybrać, aby strona fizyczna wspierająca obiekt wykonawczy, do którego się odwołuje, została zablokowana i zmapowana do jej dostępnej przestrzeni adresowej; jest to wykonywane przez wywołanie z enklawy przy użyciu podobnego przepływu jak w krokach od 1 do 4.
6. Gdy strona jest już zmapowana, silnik VTL 1 może ją bezpośrednio odczytać i sprawdzić bit ochrony obiektu procesu wykonawczego w stosunku do jego wewnętrznego kontekstu.
7. Logika VTL 1 ponownie wywołuje VTL 0, aby odwinąć mapowanie strony i odniesienie do obiektu jądra.

Raporty i ustanowienie zaufania

API oparte na WinRT jest prezentowane, aby pozwalać stronom ufającym uzyskać certyfikat sesji SGRA i podpisane raporty sesji i raporty wykonawcze. To API nie jest publiczne i jest dostępne pod NDA dla dostawców, którzy są częścią Microsoft Virus Initiative (zauważ, że Microsoft Defender Advanced Threat Protection jest obecnie jedynym komponentem wbudowanym, który łączy się bezpośrednio z SGRA poprzez to API).

Przepływ dla uzyskania zaufanego oświadczenia z SGRA jest następujący:

1. Tworzona jest sesja pomiędzy stroną ufającą a SGRA. Ustanowienie sesji wymaga połączenia sieciowego. Silnik oceniania SgrmEnclave (działający w VTL 1) generuje parę kluczy publiczny-prywatny, a chroniony proces SgrmBroker pobiera dziennik TCG i raport atestacyjny VBS, wysyłając je do usługi atestacyjnej Microsoft System Guard z publicznym składnikiem klucza wygenerowanego w poprzednim kroku.
2. Usługa atestacji weryfikuje dziennik TCG (od TPM) oraz raport atestacji VBS (jako dowód, że logika działa w enklawie VBS) i generuje raport sesji opisujący atestowane właściwości czasu rozruchu urządzenia. Podpisuje klucz publiczny kluczem pośrednim usługi atestacyjnej SGRA, aby stworzyć certyfikat, który będzie używany do weryfikacji raportów wykonawczych.

3. Raport sesji i certyfikat są zwracane stronie ufającej. Od tego momentu strona ufająca może zweryfikować ważność raportu z sesji i certyfikatu wykonawczego.
4. Okresowo, strona ufająca może zażądać od atestacji SGRA raportu wykonawczego, korzystając z ustalonej sesji: silnik oceniania SgrmEnclave generuje raport wykonawczy, opisujący stan ocen, które zostały uruchomione. Raport zostanie podpisany przy użyciu sparowanego klucza prywatnego wygenerowanego podczas tworzenia sesji i zwrócony stronie ufającej (klucz prywatny nigdy nie opuszcza enklawy).
5. Strona ufająca może zweryfikować ważność raportu wykonawczego względem uzyskanego wcześniej certyfikatu wykonawczego i podjąć decyzję o polityce opartej zarówno na zawartości raportu sesji (stan atestowany podczas rozruchu), jak i raportu wykonawczego (stan oceniony).

SGRA dostarcza pewne API, którego strony ufające mogą użyć do poświadczenia stanu urządzenia w danym momencie. API zwraca raport wykonawczy, który wyszczególnia roszczenia, że atestacja wykonawcza strażnika systemu (Windows Defender System Guard) poświadcza o stanie bezpieczeństwa systemu. Te stwierdzenia obejmują oceny, które są pomiarami wykonywanymi podczas pracy wrażliwych właściwości systemu. Na przykład aplikacja może poprosić Windows Defender System Guard o zmierzenie bezpieczeństwa systemu z enklawy wspieranej sprzętowo i zwrócić raport. Szczegóły w tym raporcie mogą być wykorzystane przez aplikację do podjęcia decyzji, czy wykonuje ona wrażliwą transakcję finansową lub wyświetla informacje osobiste.

Jak omówiono w poprzedniej sekcji, enklawa oparta na VBS może również ujawnić raport atestacyjny enklawy podpisany przez klucz określony dla VBS. Jeśli Windows Defender System Guard może uzyskać dowód, że system hosta jest uruchomiony z aktywnym VSM, może użyć tego dowodu z podpisanym raportem sesji, aby zapewnić, że konkretna enklawa jest uruchomiona. Ustanowienie zaufania niezbędnego do zagwarantowania, że raport wykonawczy jest autentyczny, wymaga zatem następujących czynności:

1. atestacji stanu rozruchu (*boot*) maszyny; binaria systemu operacyjnego, hipernadzorcy i bezpiecznego jądra muszą być podpisane przez Microsoft i skonfigurowane zgodnie z polityką bezpieczeństwa;
2. wiązania zaufania między TPM a stanem zdrowia hipernadzorcy, aby umożliwić zaufanie do dziennika pomiarów rozruchu (MBL — ang. *Measured Boot Log*);
3. wyodrębnienia potrzebnych kluczy (klucze IDK wirtualnego trybu bezpiecznego) z dziennika MBL i użycia ich do weryfikacji podpisu enklawy VBS (więcej szczegółów w rozdziale 12.);
4. podpisania publicznego składnika efemerycznej pary kluczy wygenerowanej w obrębie enklawy za pomocą zaufanego organu certyfikacji w celu wydania certyfikatu sesji;
5. podpisania raportu wykonawczego za pomocą efemerycznego klucza prywatnego.

Połączenia sieciowe pomiędzy enklawą a usługą atestacyjną systemu Windows Defender System Guard są wykonywane z poziomu VTL 0. Jednak konstrukcja protokołu atestacyjnego zapewnia, że jest on odporny na manipulacje nawet przez niezaufane mechanizmy transportowe.

Zanim opisany wcześniej łańcuch zaufania będzie mógł być wystarczająco ustanowiony, wymagane jest uruchomienie licznych technologii bazowych. Aby poinformować stronę ufającą o poziomie zaufania do raportu wykonawczego, którego mogą oczekiwać w dowolnej konkretnej konfiguracji, poziom bezpieczeństwa jest przypisywany do każdego raportu sesji podpisanego

przez usługę atestacji systemu Windows Defender System Guard. Poziom bezpieczeństwa odzwierciedla bazowe technologie włączone na platformie i przypisuje poziom zaufania oparty na możliwościach platformy. Microsoft mapuje włączanie różnych technologii zabezpieczeń do poziomów zabezpieczeń i udostępnia je, gdy interfejs API zostanie opublikowany do użytku przez firmy trzecie. Najwyższy poziom zaufania będzie prawdopodobnie wymagał co najmniej następujących cech:

- sprzętu obsługującego VBS i konfigurację OEM;
- dynamicznych pomiarów drzewa zaufania podczas uruchamiania systemu;
- bezpiecznego startu w celu weryfikacji obrazów hipernadzorcy, NT i SK;
- bezpiecznej polityki zapewniającej integralność kodu HVCI i integralność kodu w trybie jądra KMCI; podpisywanie testowe jest wyłączone oraz debugowanie jądra jest wyłączone;
- sterownika ELAM zainstalowanego w systemie.

Podsumowanie

System Windows jest w stanie zarządzać wieloma maszynami wirtualnymi i uruchamiać je dzięki hipernadzorcy Hyper-V i jego stosowi wirtualizacji, które połączone razem obsługują różne systemy operacyjne działające w maszynie wirtualnej. Z biegiem lat te dwa komponenty ewoluowały, aby zapewnić więcej optymalizacji i zaawansowanych funkcji dla maszyn wirtualnych, takich jak wirtualizacja zagnieżdżona, wiele harmonogramów dla procesorów wirtualnych, różne typy obsługi sprzętu wirtualnego, VMBus, maszyny wirtualne ze wsparciem VA itd.

Bezpieczeństwo oparte na wirtualizacji zapewnia głównemu systemowi operacyjnemu nowy poziom ochrony przed złośliwym oprogramowaniem i rootkitami ze zdolnościami do wykradania danych, które nie są już w stanie wykraść prywatnych i poufnych informacji z pamięci głównego systemu operacyjnego. Bezpieczne jądro wykorzystuje usługi dostarczane przez hipernadzorcę systemu Windows do tworzenia nowego środowiska wykonawczego (VTL 1), które jest chronione i niedostępne dla oprogramowania uruchomionego w głównym systemie operacyjnym. Ponadto bezpieczne jądro dostarcza wiele usług do ekosystemu Windows, które pomagają utrzymać bardziej bezpieczne środowisko.

Bezpieczne jądro definiuje również tryb izolowanego użytkownika (IUM), pozwalając na wykonywanie kodu trybu użytkownika w nowym, chronionym środowisku poprzez trustlety, bezpieczne urządzenia i enklawy. Rozdział zakończył się analizą atestacji wykonawczej strażnika systemu (SGRA), komponentu, który wykorzystuje usługi prezentowane przez bezpieczne jądro do pomiaru środowiska wykonawczego stacji roboczej i zapewnienia silnych gwarancji jego integralności.

W następnym rozdziale przyjrzymy się komponentom zarządzania i diagnostyki systemu Windows i omówimy ważne mechanizmy związane z ich infrastrukturą: rejestr, usługi, Harmonogram zadań, WMI, śledzenie zdarzeń jądra itd.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

Z

Ż

...a więc uważasz, że *dobrze* znasz Windowsa?

Windows 3.1 rozpoczął rewolucję w świecie komputerów; od tej pory Microsoft wydał wiele generacji „okienek". Dziś Windows jest dojrzałym, bezpiecznym, niezawodnym i skalowalnym systemem. Aby w pełni wykorzystać ten potencjał, trzeba dobrze zrozumieć, jak funkcjonują podstawowe wewnętrzne komponenty systemu, jakie są zasady rządzące ich wydajnością, a także czym się charakteryzują funkcje bezpieczeństwa nowoczesnych systemów Windows.

W drugiej części tego zaktualizowanego przewodnika dla zaawansowanych informatyków znalazł się między innymi opis mechanizmu wywołania ALPC i procedury synchronizacji sterowników urządzeń i aplikacji. Omówiono zasady wirtualizacji, a także takie elementy jak rejestr, interfejs WMI, usługi ETW i DTrace. Pokazano najważniejsze komponenty pamięci masowej i systemy plików, w tym NTFS i ReFS. Zaprezentowano też operacje zachodzące podczas rozruchu i zamykania systemu. Uwzględniono aktualizacje 21H1/2104 systemu Windows 10, jak również systemów Windows Server 2022, 2019 i 2016. Dodatkowo ujęto tu obszerne wprowadzenie do platformy Hyper-V.

Dzięki temu przewodnikowi możesz:

- zrozumieć wirtualizację systemu i działanie opartych na niej zabezpieczeń
- zdobyć wiedzę o mechanizmach zarządzania kluczami, rejestrem, usługami Windows
- poznać operacje wykonywane podczas rozruchu systemu Windows
- przeanalizować bezpieczny rozruch oparty na interfejsie UEFI

Andrea Allievi specjalizuje się w inżynierii wstecznej i w zaawansowanych zagadnieniach systemów operacyjnych. Napisał wiele systemowych narzędzi do usuwania szkodliwego kodu. Obecnie pracuje w zespole Kernel Security Core w Microsofcie.

Alex Ionescu jest wybitnym architektem bezpieczeństwa oprogramowania i ekspertem w dziedzinie niskopoziomowego oprogramowania systemowego. Obecnie pełni funkcję wiceprezesa spółki CrowdStrike.

Mark E. Russinovich jest współtwórcą oprogramowania Sysinternals. Obecnie pracuje jako dyrektor do spraw technologii w Azure i współpracownik do spraw technicznych w Microsofcie. Jest uznanym ekspertem w dziedzinie systemów operacyjnych i bezpieczeństwa.

David A. Solomon przez 20 lat prowadził szkolenia z zakresu wewnętrznych komponentów systemu Windows. W latach 1993 i 2005 uzyskał certyfikaty Microsoft Support Most Valuable Professional (MVP).

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9072-0

9 788328 390720

Cena: 179,00 zł