HyperGuard – Secure Kernel Patch Guard: Part 1 – SKPG Initialization

windows-internals.com/hyperguard-secure-kernel-patch-guard-part-1-skpg-initialization

By Yarden Shafir

This will be a multi-part series of posts describing the internal mechanisms and purpose of Secure Kernel Patch Guard, also known as HyperGuard. This first part will focus on what SKPG is and how it's being initialized.

Overview

In the world of Windows security, PatchGuard is a uniquely undocumented and hardly any "unofficial" documentation. Thus, there are conflicting opinions and rumors about the way it operates and different "PatchGuard bypasses" that get published aren't very reliable. Still, every few years some helpful PG analysis gets published, shedding some light on this mysterious feature. This blog post is not about PatchGuard so we won't go into much detail about it, but it discusses a similar and related feature, so some basic knowledge of PatchGuard is needed. Here are a couple of things needed to understand of the rest of the post:

- The purpose of PatchGuard is to monitor the system for changes in kernel space that should not happen on a normal system and crash it when those are detected. This doesn't mean any unusual data change – PatchGuard monitors a pre-determined list of data structures that are common targets for kernel exploitation or rootkits, such as modifications to `HalDispatchTable` or callback arrays, or changes to control registers or MSRs to disable security features. The full list of monitored structures and pointers is not documented and the information that does get published by Microsoft is left vague on purpose.
- PatchGuard doesn't monitor everything, all the time. It runs periodically, checking for certain changes every time it runs – it won't necessarily crash the system right when a malicious change is done and a system might run for a long time with such changes. There is no guarantee that PatchGuard will ever detect and crash the system. This also means it is hard to validate potential bypasses.

The main weakness of PatchGuard and the reason for all the obscurity around its implementation is the fact that it monitors Ring `0` code and data – from code that runs in Ring `0`. There is nothing preventing a rootkit that already gained Ring `0` code execution privileges from patching the code for PatchGuard itself and disabling or bypassing it. The only thing stopping this scenario is PatchGuard's obscurity and the fact that its code is hard to find and uses a range of obfuscation techniques to make itself hard to analyze and disable.

There is a lot more to say about PatchGuard but, like I mentioned, this is not the topic of the post. So, I'll skip right to discussing PatchGuard's newer sibling – HyperGuard, also known as Secure Kernel Patch Guard, or `SKPG`. This new feature leverages the existence of Hyper-V and VBS to create a new monitoring and protection capability that is similar to PatchGuard but not susceptible to the same weaknesses since it is not running as normal Ring `0` code and cannot be tampered by normal rootkits.

Finding HyperGuard

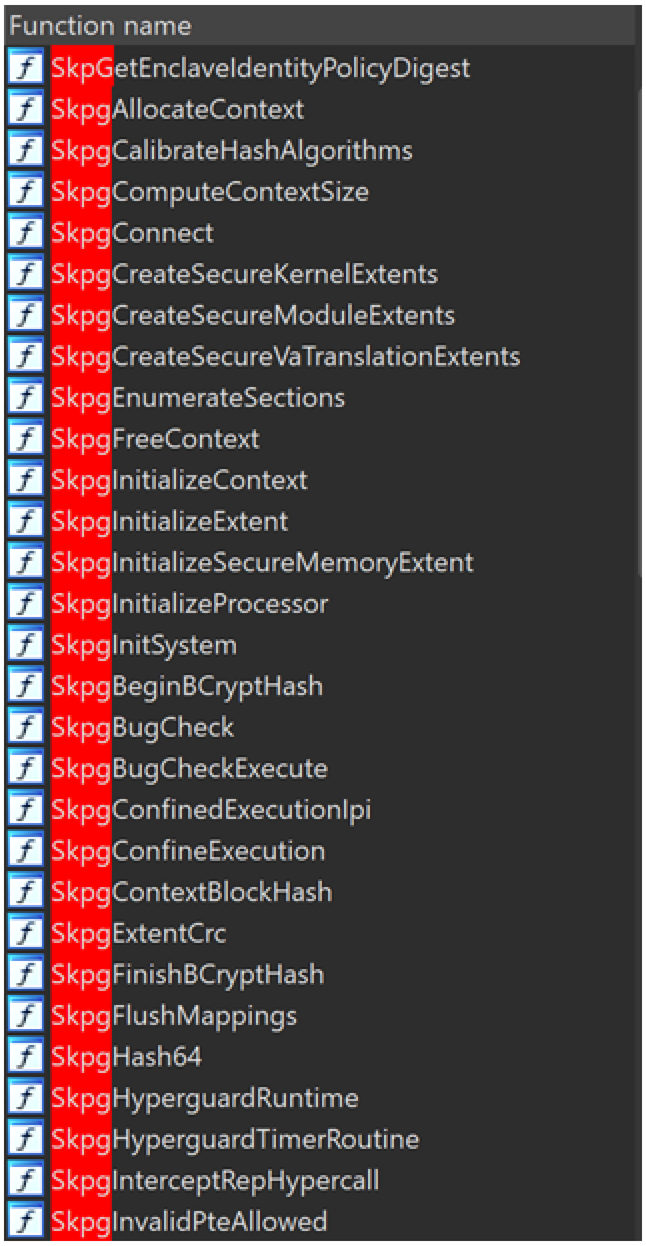
HyperGuard takes advantage of `VBS` – Virtualization Based Security. This capability that was added in the past few years is made possible by the creation of Hyper-V and Virtual Trust Levels (`VTL` s). The hypervisor allows creating a system where most things run in `VTL0`, but some, more privileged things, run in higher `VTL` s (currently the only one implemented is `VTL1`) where they are not accessible to normal processes regardless of their privilege level – including `VTL0` kernel code. Put simply, no `VTL0` code can interact with memory in `VTL1` in any way.

Having memory that cannot be tampered with even from normal kernel code allows for many new security features, some of which I've written about in the past and others are documented in other blogs, conference talks and official Microsoft documentation. A few examples include `KCFG`, `HVCI` and `KDP`.

This is also what allows Microsoft to implement `HyperGuard` – a feature similar to `PatchGuard` that can't be tampered with even by malicious code that managed to elevate itself to run in the kernel. For this reason, `HyperGuard` doesn't need to hide or obfuscate itself in any way, and it's so much easier to analyze using static analysis tools.

The `VTL1` kernel, also known as the secure kernel, is managed through `SecureKernel.exe`. This is also the binary where `HyperGuard` is implemented. If we open `securekernel.exe` in IDA we can easily find all the code implementing `HyperGuard`, which all uses the prefix `Skpg`:

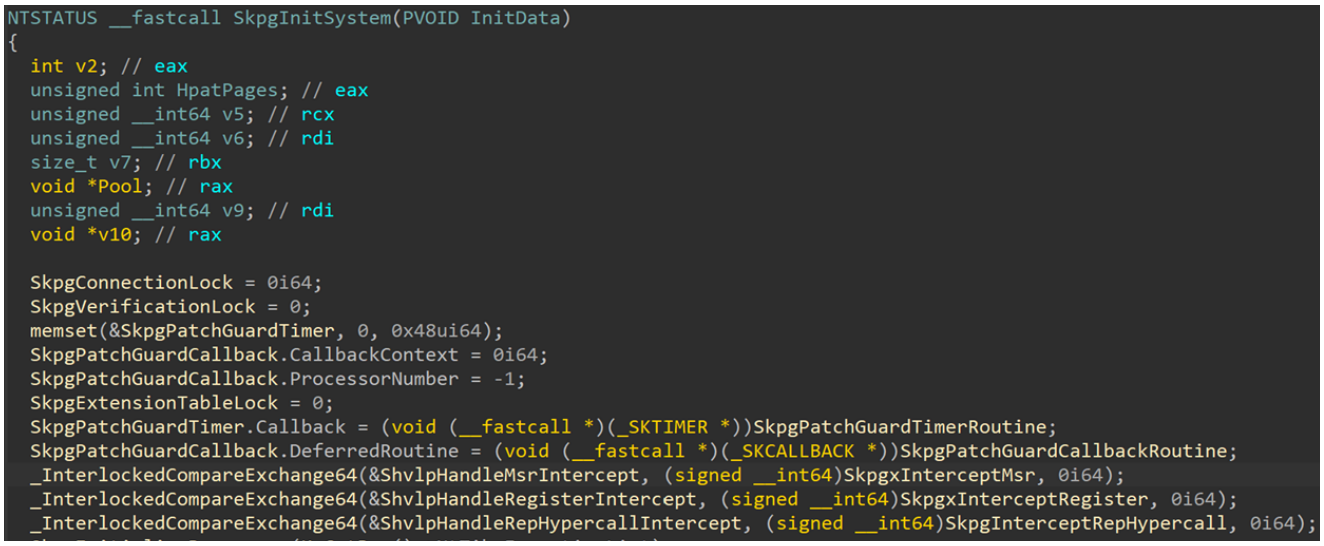
This series will cover some of those functions, starting from the first ones being called during boot: `SkpgInitSystem`:



```
Function name
SkpGetEnclaveIdentityPolicyDigest
SkpgAllocateContext
SkpgCalibrateHashAlgorithms
SkpgComputeContextSize
SkpgConnect
SkpgCreateSecureKernelExtents
SkpgCreateSecureModuleExtents
SkpgCreateSecureVaTranslationExtents
SkpgEnumerateSections
SkpgFreeContext
SkpgInitializeContext
SkpgInitializeExtent
SkpgInitializeSecureMemoryExtent
SkpgInitializeProcessor
SkpgInitSystem
SkpgBeginBCryptHash
SkpgBugCheck
SkpgBugCheckExecute
SkpgConfinedExecutionIpi
SkpgConfineExecution
SkpgContextBlockHash
SkpgExtentCrc
SkpgFinishBCryptHash
SkpgFlushMappings
SkpgHash64
SkpgHyperguardRuntime
SkpgHyperguardTimerRoutine
SkpgInterceptRepHypercall
SkpgInvalidPteAllowed
```

HyperGuard Initialization

`HyperGuard` initialization mostly happens during the normal kernel's Phase `1` initialization, but requires multiple steps. The first step starts with a secure call where `SKSERVICE=SECURESERVICE_PHASE3_INIT`. This leads to `SkInitSystem` which will initialize `SKCI` (Secure Kernel Code Integrity) and call into `SkpgInitSystem`. This function sets up the basic components of `SKPG` – its callback, timer, extension table and intercept functions, all of which I'll discuss in more detail later in this series. At this point `SKPG` is not fully initialized – that only happens later in response to another request from the normal kernel. For now, only a few `SKPG` globals are being set:



```
NTSTATUS __fastcall SkpgInitSystem(PVOID InitData)
{
  int v2; // eax
  unsigned int HpatPages; // eax
  unsigned __int64 v5; // rcx
  unsigned __int64 v6; // rdi
  size_t v7; // rbx
  void *Pool; // rax
  unsigned __int64 v9; // rdi
  void *v10; // rax

  SkpgConnectionLock = 0i64;
  SkpgVerificationLock = 0;
  memset(&SkpgPatchGuardTimer, 0, 0x48ui64);
  SkpgPatchGuardCallback.CallbackContext = 0i64;
  SkpgPatchGuardCallback.ProcessorNumber = -1;
  SkpgExtensionTableLock = 0;
  SkpgPatchGuardTimer.Callback = (void (__fastcall *)(_SKTIMER *))SkpgPatchGuardTimerRoutine;
  SkpgPatchGuardCallback.DeferredRoutine = (void (__fastcall *)(_SKCALLBACK *))SkpgPatchGuardCallbackRoutine;
  _InterlockedCompareExchange64(&ShvlpHandleMsrIntercept, (signed __int64)SkpgxInterceptMsr, 0i64);
  _InterlockedCompareExchange64(&ShvlpHandleRegisterIntercept, (signed __int64)SkpgxInterceptRegister, 0i64);
  _InterlockedCompareExchange64(&ShvlpHandleRepHypercallIntercept, (signed __int64)SkpgInterceptRepHypercall, 0i64);
```

Some interesting components to notice at this stage are:

- `SkpgPatchGuardCallback` – a callback which is going to be called every time `HyperGuard` checks need to run and will invoke the target function `SkpgPatchGuardCallbackRoutine`.
- `SkpgPatchGuardTimer` – a secure kernel timer object that is going to control the execution of some `HyperGuard` checks. It gets set to run at a random time so checks will happen at different intervals, making periodic checks harder to avoid. The function set its callback function to `SkpgPatchGuardTimerRoutine`.
- Intercept function pointers – other than the periodic checks controlled by the timer, `HyperGuard` also has a few intercept functions, which execute every time a certain operation is being intercepted by the Hypervisor. The operation being intercepted is pretty clear from the function names, but I'll cover them in more detail later anyway. The global variables for these are:
  - `ShvlpHandleMsrIntercept` – points to `SkpgxInterceptMsr`
  - `ShvlpHandleRegisterIntercept` – points to `SkpgxInterceptRegister`
  - `ShvlpHandleRepHypercallIntercept` – points to `SkpgInterceptRepHypercall`
- Optional variables – there are a few other global variables that did not fit in the screenshot and get initialized based on the flags received as part of the input argument, or other optional configuration:
  - `SkpgInhibitKernelVaProtection`
  - `SkpgNtKvaShadow`
  - `SkpgSecureExtension`

After initializing all the global variables, the function returns and the rest of the secure kernel initialization continues. For now, the timer is not scheduled and `HyperGuard` is effectively "dormant". `HyperGuard` is only fully "activated" later – through a call to `SkpgConnect`.

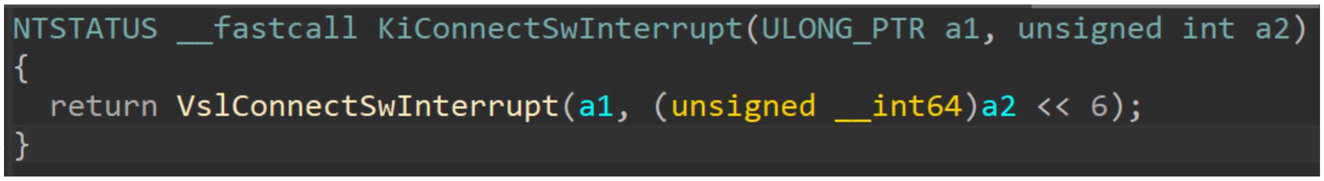
There are three ways to call SkpgConnect and all start from a call by the normal kernel:

HyperGuard Activation

Connect Software Interrupt – the PatchGuard Path

The most interesting `HyperGuard` activation path is through `PatchGuard`. This `SKPG` activation path, like all others, begins with a secure call. This secure call, with `SKSERVICE=SECURESERVICE_CONNECT_SW_INTERRUPT`, originates from the normal kernel function `VslConnectSwInterrupt`. This leads, as usual, to the secure kernel handler which calls into `IumpConnectSwInterrupt` and from there to `SkpgConnect`, passing it all the data that was sent by the normal kernel.

When we search for calls to `VslConnectSwInterrupt` we see two calls – one from `PsNotifyCoreDriversInitialized` that I'll cover soon and a second one from `KiConnectSwInterrupt`:



```
NTSTATUS __fastcall KiConnectSwInterrupt(ULONG_PTR a1, unsigned int a2)
{
  return VslConnectSwInterrupt(a1, (unsigned __int64)a2 << 6);
}
```

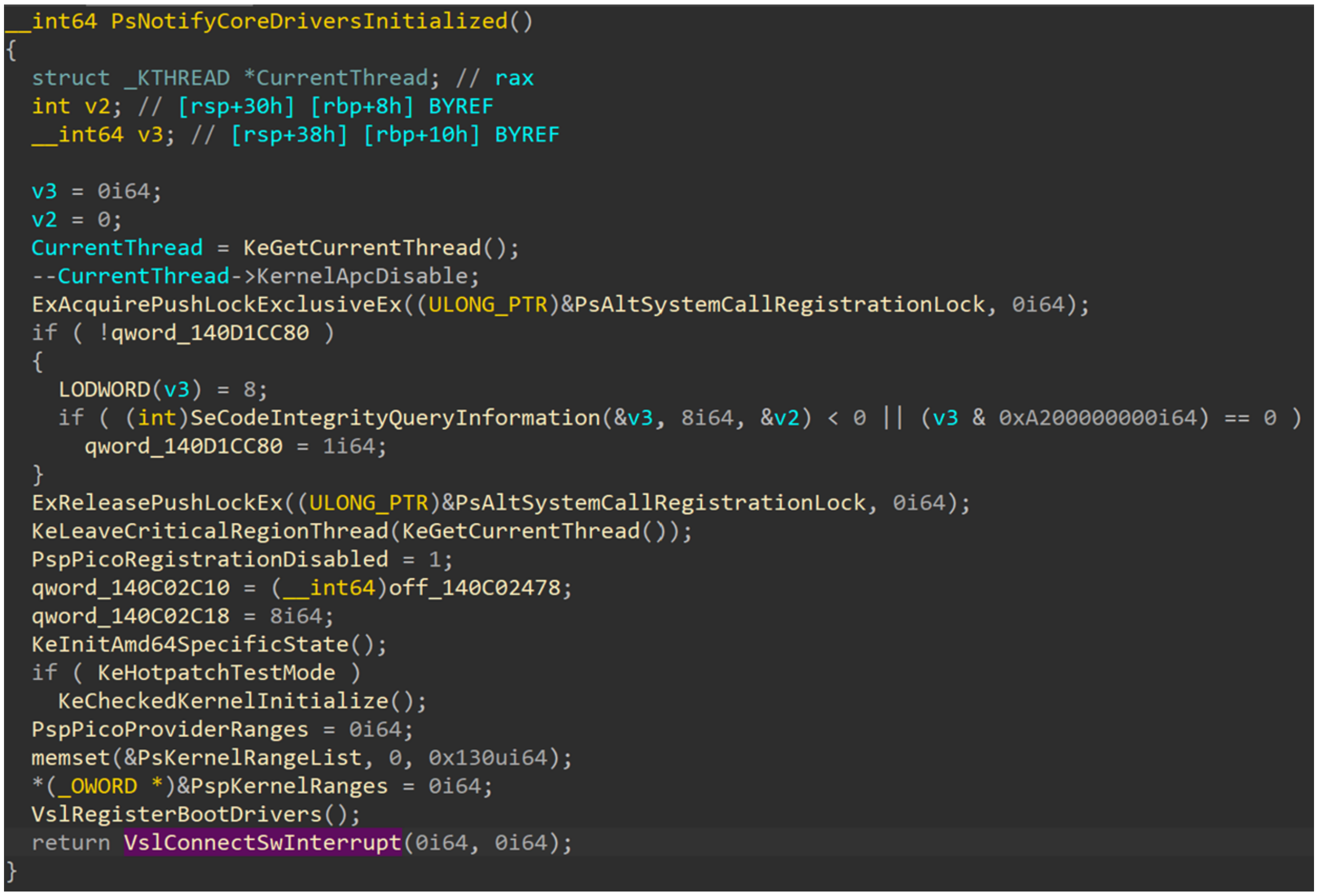
`KiConnectSwInterrupt` is only called by one caller – an anonymous function in `ntoskrnl.exe` that has no name in the public symbols. This is an extremely large function that calls into other anonymous functions and has a lot of weird and seemingly unrelated functionality. This is one of the `PatchGuard` initialization routines, which does the "real" activation of `HyperGuard`, supplying the secure kernel with memory protection ranges and targets which I will discuss later when talking about `SKPG` extents.

I encourage you to follow the call stack yourselves and get a bit of insight into the mysteries of `PatchGuard` initialization, but if I start covering `PatchGuard` details this series will quickly become a book so I will skip the details here. Let's just trust me when I say that this all also happens in the context of Phase `1` initialization and is the first point where `HyperGuard` is activated.

Once `HyperGuard` is fully activated, a global variable `SkpgInitialized` is set to `TRUE`. This variable is checked every time `SkpgConnect` is called, and if set the function will return immediately and not make any changes to any `SKPG` initialization data. This means that the two other activation paths that will be described here will only activate `HyperGuard` if `PatchGuard` is not running and will result in less thorough protection of the machine. If `PatchGuard` is active, then the other two activation paths will return without doing anything.

Connect Software Interrupt – Phase1 Initialization

The second code path into `VslConnectSwInterrupt` goes through `PsNotifyCoreDriversInitialized`. This is also happening as part of Phase `1` initialization, but later than the `PatchGuard` path:



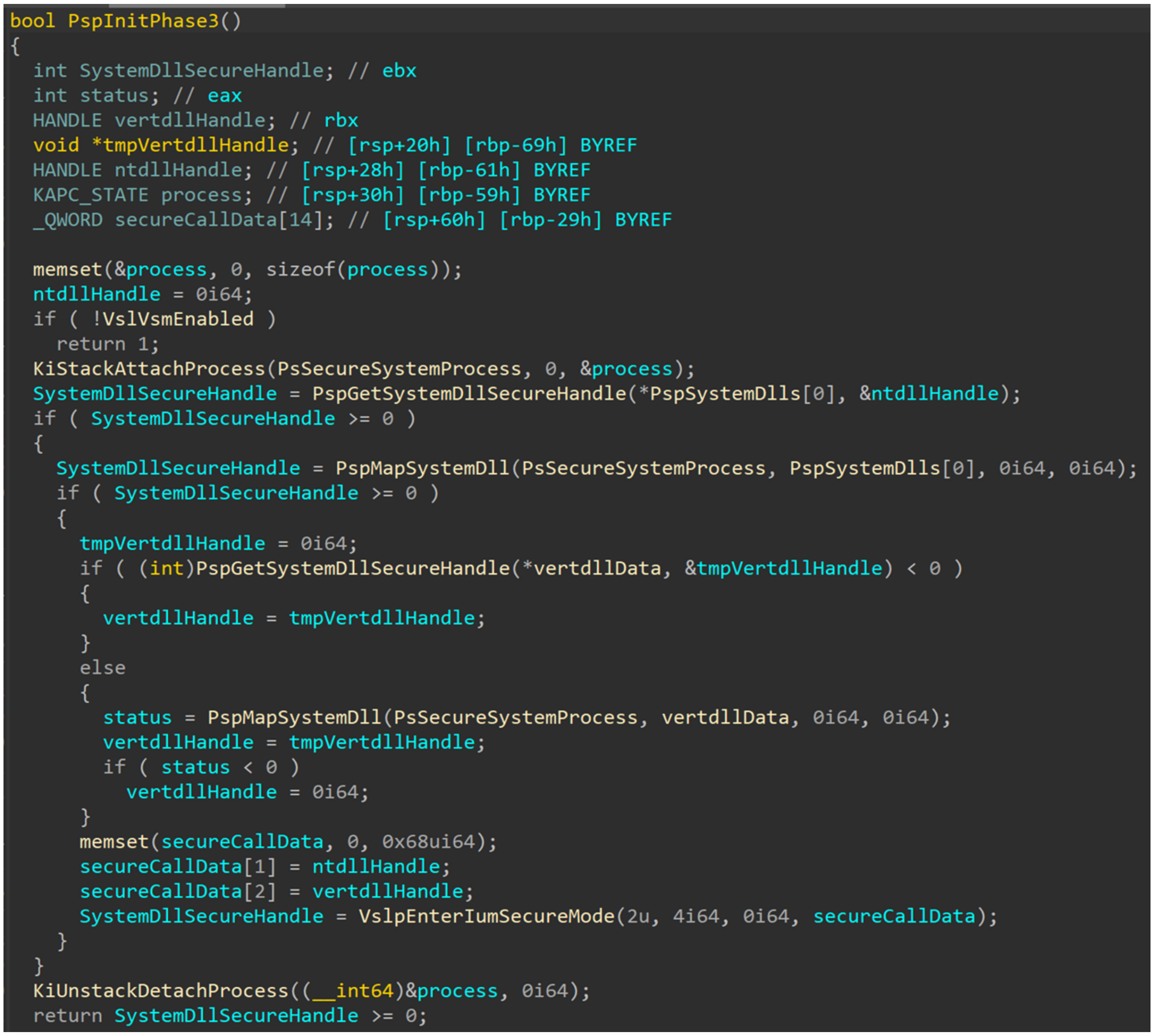
```
__int64 PsNotifyCoreDriversInitialized()
{
  struct _KTHREAD *CurrentThread; // rax
  int v2; // [rsp+30h] [rbp+8h] BYREF
  __int64 v3; // [rsp+38h] [rbp+10h] BYREF

  v3 = 0i64;
  v2 = 0;
  CurrentThread = KeGetCurrentThread();
  --CurrentThread->KernelApcDisable;
  ExAcquirePushLockExclusiveEx((ULONG_PTR)&PsAltSystemCallRegistrationLock, 0i64);
  if ( !qword_140D1CC80 )
  {
    LODWORD(v3) = 8;
    if ( (int)SeCodeIntegrityQueryInformation(&v3, 8i64, &v2) < 0 || (v3 & 0xA2000000000i64) == 0 )
      qword_140D1CC80 = 1i64;
  }
  ExReleasePushLockEx((ULONG_PTR)&PsAltSystemCallRegistrationLock, 0i64);
  KeLeaveCriticalRegionThread(KeGetCurrentThread());
  PspPicoRegistrationDisabled = 1;
  qword_140C02C10 = (__int64)off_140C02478;
  qword_140C02C18 = 8i64;
  KeInitAmd64SpecificState();
  if ( KeHotpatchTestMode )
    KeCheckedKernelInitialize();
  PspPicoProviderRanges = 0i64;
  memset(&PsKernelRangeList, 0, 0x130ui64);
  *(_OWORD *)&PspKernelRanges = 0i64;
  VslRegisterBootDrivers();
  return VslConnectSwInterrupt(0i64, 0i64);
}
```

As we can see here, the call to `VslConnectSwInterrupt` is done with empty input variables, meaning no memory ranges or extra data is sent to `HyperGuard` and it will only use its basic functionality. If `PatchGuard` is running, then at this point `SKPG` should already be initialized and the call will return with no changes to `SKPG`, so this path is only needed if `PatchGuard` is not active.

Phase3 Initialization

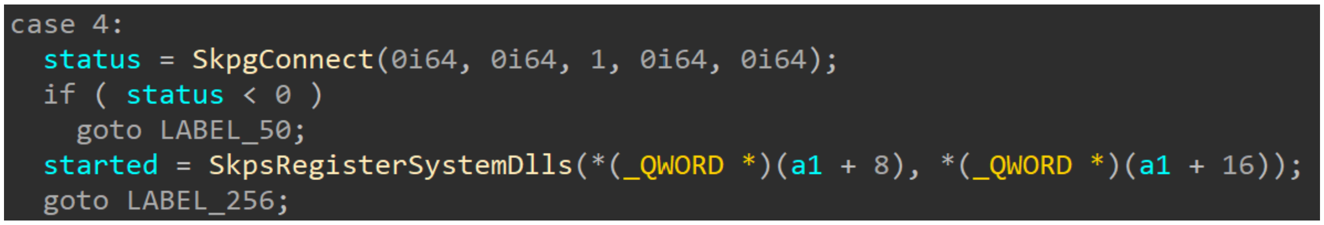
The last case where `HyperGuard` is activated happens during Phase `3` initialization. This happens in response to a secure call with `SKSERVICE=SECURESERVICE_REGISTER_SYSTEM_DLLS`. It will also call into `SkpgConnect` with no input data, simply to initialize it if nothing else has already.

On the normal kernel side: In `PspInitPhase3` the system checks the `VslVsmEnabled` global variable to learn whether Hyper-V is running and `VSM` is enabled. If it is, the system calls `VslpEnterIumSecureMode` – a common function to generate a secure call with a given service code and arguments packed into an `MDL`. The system enters secure mode with service code `SECURESERVICE_REGISTER_SYSTEM_DLLS`:



```
bool PspInitPhase3()
{
  int SystemDllSecureHandle; // ebx
  int status; // eax
  HANDLE vertdllHandle; // rbx
  void *tmpVertdllHandle; // [rsp+20h] [rbp-69h] BYREF
  HANDLE ntdllHandle; // [rsp+28h] [rbp-61h] BYREF
  KAPC_STATE process; // [rsp+30h] [rbp-59h] BYREF
  _QWORD secureCallData[14]; // [rsp+60h] [rbp-29h] BYREF

  memset(&process, 0, sizeof(process));
  ntdllHandle = 0i64;
  if ( !VslVsmEnabled )
    return 1;
  KiStackAttachProcess(PsSecureSystemProcess, 0, &process);
  SystemDllSecureHandle = PspGetSystemDllSecureHandle(*PspSystemDlls[0], &ntdllHandle);
  if ( SystemDllSecureHandle >= 0 )
  {
    SystemDllSecureHandle = PspMapSystemDll(PsSecureSystemProcess, PspSystemDlls[0], 0i64, 0i64);
    if ( SystemDllSecureHandle >= 0 )
    {
      tmpVertdllHandle = 0i64;
      if ( (int)PspGetSystemDllSecureHandle(*vertdllData, &tmpVertdllHandle) < 0 )
      {
        vertdllHandle = tmpVertdllHandle;
      }
      else
      {
        status = PspMapSystemDll(PsSecureSystemProcess, vertdllData, 0i64, 0i64);
        vertdllHandle = tmpVertdllHandle;
        if ( status < 0 )
          vertdllHandle = 0i64;
      }
      memset(secureCallData, 0, 0x68ui64);
      secureCallData[1] = ntdllHandle;
      secureCallData[2] = vertdllHandle;
      SystemDllSecureHandle = VslpEnterIumSecureMode(2u, 4i64, 0i64, secureCallData);
    }
  }
  KiUnstackDetachProcess((__int64)&process, 0i64);
  return SystemDllSecureHandle >= 0;
```

Once a secure call reaches the secure kernel it is handled by `IumInvokeSecureService`, which is pretty much just a big switch statement, calling the correct function or functions for each service code. In the case of code `SECURESERVICE_REGISTER_SYSTEM_DLLS`, it calls `SkpgConnect` and then uses the data passed in by the kernel to register system `DLL`s:

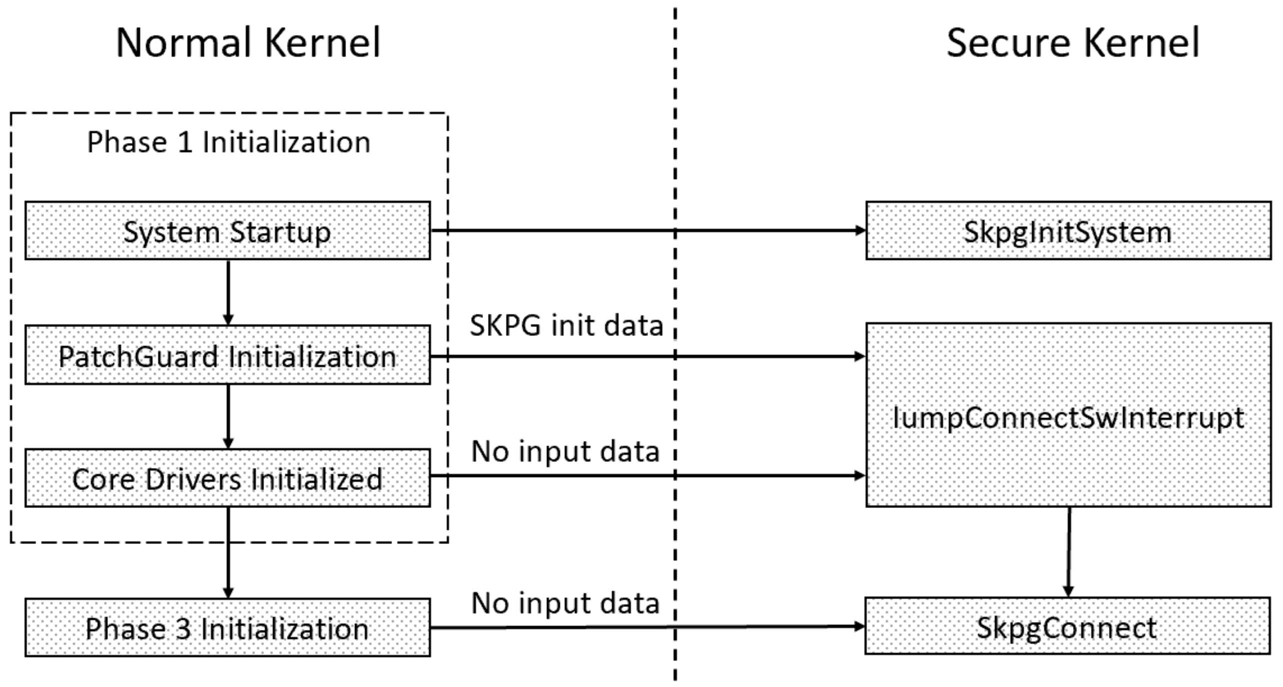


```
case 4:
  status = SkpgConnect(0i64, 0i64, 1, 0i64, 0i64);
  if ( status < 0 )
    goto LABEL_50;
  started = SkpsRegisterSystemDlls(*(_QWORD *)(a1 + 8), *(_QWORD *)(a1 + 16));
  goto LABEL_256;
```

As I mentioned, this is the last time `SkpgConnect` is called, right at the end of system initialization. This is done in case `SKPG` hasn't been initialized at an earlier stage already. In this case, `SkpgConnect` is called with almost no input data, to only initialize the most basic

`SKPG` functionality. If `SKPG` has already been initialized earlier, this call will return without changing anything.

HyperGuard Activation – Diagram



This is it for part `1` of this series. So far, we only covered the general idea of what `HyperGuard` is and its initialization paths. Next time we will dive into `SkpgConnect` to see what happens during `SKPG` activation and learn more about the types of data `SKPG` protects and how.