HyperGuard Part 3 – More SKPG Extents

windows-internals.com/hyperguard-part-3-more-skpg-extents

By Yarden Shafir

Hi all! And welcome to part 3 of the HyperGuard chronicles!

In the previous blog post I introduced SKPG extents – the data structures that describe the memory ranges and system components that should be monitored by HyperGuard. So far, I only covered the initialization extent and various types of memory extents, but those are just the beginning. In this post I will cover the rest of the extent types and show how they are used by HyperGuard to protect other areas of the system.

The next extent group to look into is MSR and Control Register extents:

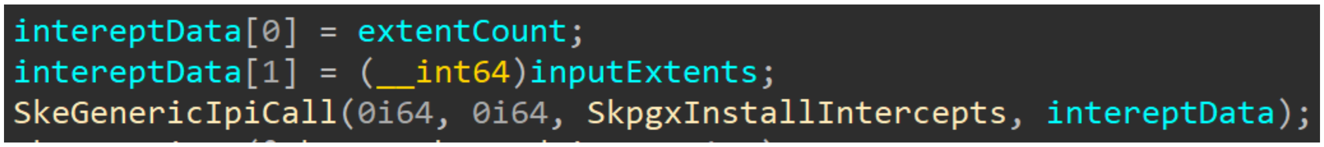
MSR and Control Register Extents

This group contains the following extent types:

- `0x1003` : `SkpgExtentMsr`
- `0x1006` : `SkpgExtentControlRegister`
- `0x100C` : `SkpgExtentExtendedControlRegister`

These extent types are received from the normal kernel, but they are never added into the array at the end of the `SKPG_CONTEXT` or get validated during the runtime checks that I'll describe in one of the next posts. Instead, they are used in yet another part of SKPG initialization.

After initializing the `SKPG_CONTEXT` in `SkpgInitializeContext` , `SkpgConnect` performs an IPI (Inter-Processor Interrupt). It performs this IPI by calling `SkeGenericIpiCall` with a target function and input data, and the function will call the target function on every processor and send the requested data. In this case, the target function is `SkpgxInstallIntercepts` and the input data contains the number of input extents and the matching array:



```
intereptData[0] = extentCount;
intereptData[1] = (__int64)inputExtents;
SkeGenericIpiCall(0i64, 0i64, SkpgxInstallIntercepts, intereptData);
```

I will go over intercepts in a lot more detail in a future blog post, but to give some necessary context: SKPG can ask the hypervisor to intercept certain actions in the system, like memory access, register access or instructions. HyperGuard uses that ability to intercept access to certain MSRs and Control Registers (and other things, which I will talk about later) to prevent malicious modifications. HyperGuard uses the input extents to choose which MSRs and Control Registers to intercept, out of the list of accepted options.

Since each processor has its own set of MSRs and registers, HyperGuard needs to intercept the requested one on all processors. Therefore, `SkpgxInstallIntercepts` is called through an IPI, to make sure it's called in the context of each processor.

Once in `SkpgxInstallIntercepts`, the function iterates over the array of input extents and handles the three types included in this group based on the data supplied in them. If you remember, each extent contains `0x18` bytes of type-specific data. For this group, this data contains the number of the MSR/Register to be intercepted as well as the processor number that it should be intercepted on. This means that there might be more that one input extent for each MSR or control register, each for a different processor number. Or MSRs and control registers might only be intercepted on certain processors but not on others, if that is what the normal kernel requested. The data structure in the input extent for MSRs and control register extents looks something like this:

```
typedef struct _MSR_CR_DATA
{
    ULONG64 Mask;
    ULONG64 Value;
    ULONG RegisterNumber;
    ULONG ProcessorNumber;
} MSR_CR_DATA, *PMSR_CR_DATA;
```

While iterating over the extents, the function checks if the extent type is of one of the three in this group, and if so whether the processor number in the extent matches the current processor. If so, it checks if the number of the MSR or control register matches one of the accepted ones. If the extent matches one of the accepted registers, a mask is fetched from an array in the `SKPRCB` – this array contains the needed masks for all accepted MSRs and control registers so the hypervisor can be asked to intercept them. All masks are collected, and when all extents have been examined the final mask is sent to `ShvlSetRegisterInterceptMasks` to be installed. The mask that is used to install the intercepts is the union `HV_REGISTER_CR_INTERCEPT_CONTROL`. It is documented and can be found here.

Now that the general process is covered, we can look into the accepted MSRs and control registers and understand why HyperGuard might want to protect them from modifications, starting from the MSRs:

SkpgExtentMsr

Patching certain MSRs is a popular operation for exploits and rootkits, allowing them to do things such as hooking system calls or disabling security features. Some of those MSRs are already periodically monitored by PatchGuard, but there are benefits to intercepting them through HyperGuard that I will cover later. The list of MSRs that can be intercepted keeps growing over time and receives new additions as new features and registers get added to CPUs, such as the implementation of CET which added multiple MSRs that might be a target for attackers. As of Windows 11 build 22598, the MSRs that can be intercepted by SKPG are:

1. `IA32_EFER` (`0xC0000080`) – among other things, this MSR contains the NX bit, enforcing a mitigation that doesn't allow code execution in addresses that aren't specifically marked as executable. It also contains flags related to virtualization support.
2. `IA32_STAR` (`0xC0000081`) – contains the address of the `x86` system call handler.
3. `IA32_LSTAR` (`0xC0000082`) – contains the address of the `x64` system call handler – should normally be pointing to `nt!KiSystemCall64` .
4. `IA32_CSTAR` (`0xC0000083`) – contains the address of the system call handler on `x64` when running in compatibility mode – should normally be pointing to `nt!KiSystemCall32` .
5. `IA32_SFMASK` (`0xC0000084`) – system call flags mask. Any bit set here when a system call is executed will be cleared from `EFLAGS` .
6. `IA32_TSC_AUX` (`0xC0000103`) – usage depends on the operating system, but this MSR is generally used to store a signature, to be read together with a time stamp.
7. `IA32_APIC_BASE` (`0x1B`) – contains the `APIC` base address.
8. `IA32_SYSENTER_CS` (`0x174`) – contains the `CS` value for ring `0` code when performing system calls with `SYSENTER` .
9. `IA32_SYSENTER_ESP` (`0x175`) – contains the stack pointer for the kernel stack when performing system calls with `SYSENTER` .
10. `IA32_SYSENTER_EIP` (`0x176`) – contains the `EIP` value for ring `0` entry when performing system calls with `SYSENTER` .
11. `IA32_MISC_ENABLE` (`0x1A0`) – controls multiple processor features, such as Fast Strings disable, performance monitoring and disable of the XD (no-execute) bit.
12. `MSR_IA32_S_CET` (`0x6A2`) – controls kernel mode `CET` setting.
13. `IA32_PL0_SSP` (`0x6A4`) – contains the ring `0` shadow stack pointer.
14. `IA32_PL1_SSP` (`0x6A5`) – contains the ring `1` shadow stack pointer.
15. `IA32_PL2_SSP` (`0x6A6`) – contains the ring `2` shadow stack pointer.
16. `IA32_INTERRUPT_SSP_TABLE_ADDR` (`0x6A8`) – contains a pointer to the interrupt shadow stack table.
17. `IA32_XSS` (`0xDA0`) – contains a mask to be used when `XSAVE` and `XRESTOR` instructions are called in kernel-mode. For example, it controls the saving and loading of the registers used by Intel Processor Trace (`IPT`).

SkpgExtentControlRegister

By modifying certain control registers an attacker can disable security features or gain control of execution. Currently SKPG supports intercepts of two control registers:

1. `CR0` – controls certain hardware configuration such as paging, protected mode and write protect.
2. `CR4` – controls the configuration of different hardware features. For example, driver signature enforcement, `SMEP` and `UMIP` bits control security features that make `CR4` an interesting target for attackers using an arbitrary write exploit.

SkpgExtentExtendedControlRegister

Currently only one extended control register exists – `XCR0`. It's used to toggle storing or loading of extended registers such as `AVX`, `ZMM` and `CET` registers, and can be intercepted and protected by SKPG.

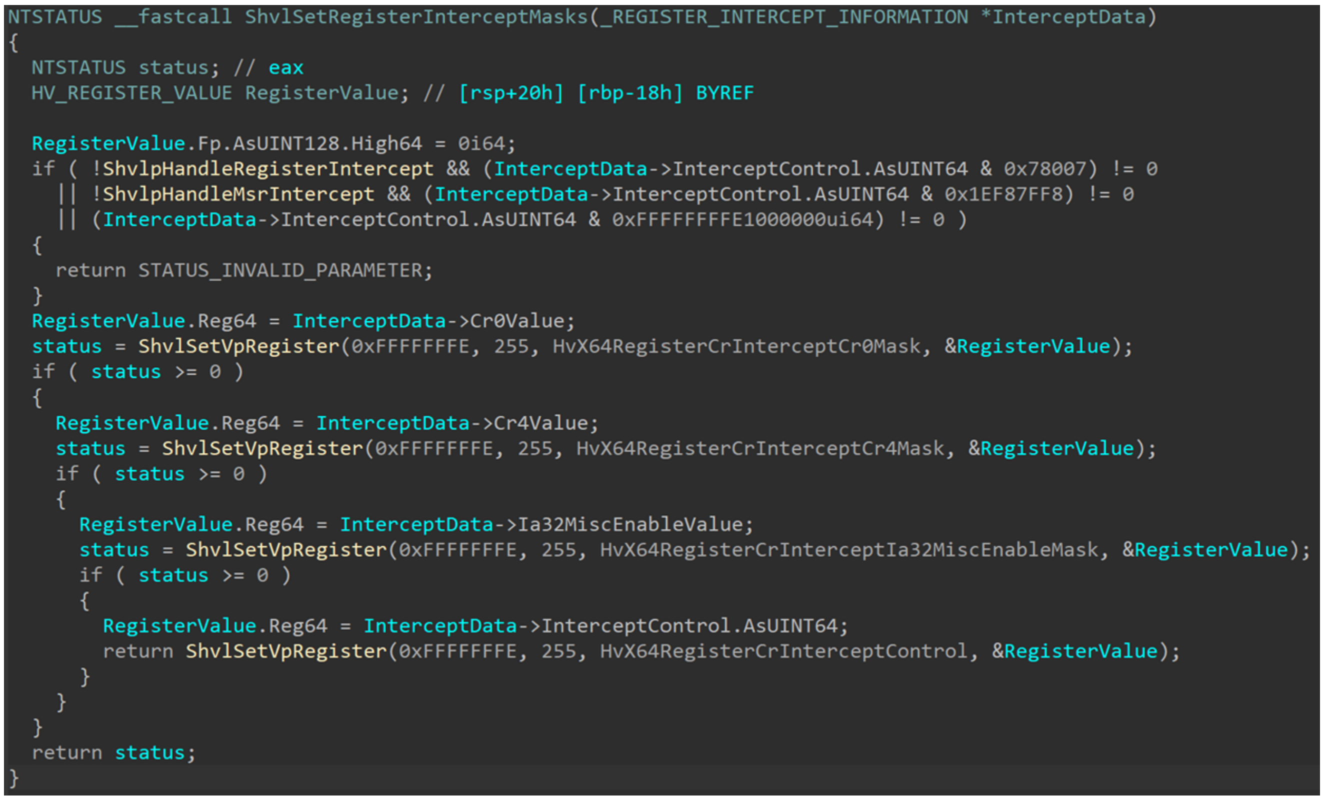
Installing the Intercepts

Now that we know that registers can be intercepted and why, we can get back and look at the installation of the intercepts through `ShvlSetRegisterInterceptMasks`. The function receives a `HV_REGISTER_CR_INTERCEPT_CONTROL` mask to know which intercepts to install, as well as the values for a few of the intercepted registers – `CR0`, `CR4` and `IA32_MISC_ENABLE` MSR. These are all placed in a structure that is passed into the function, which looks like this:

```
struct _REGISTER_INTERCEPT_INFORMATION
{
    HV_REGISTER_CR_INTERCEPT_CONTROL InterceptControl;
    ULONG64 Cr0Value;
    ULONG64 Cr4Value;
    ULONG64 Ia32MiscEnableValue;
} REGISTER_INTERCEPT_INFORMATION, *PREGISTER_INTERCEPT_INFORMATION;
```

The `InterceptControl` mask is built while iterating over the input extents, and the values for `CR0`, `CR4` and `IA32_MISC_ENABLE` are read from the `SKPRCB` (their values, together with the values for all other potentially-intercepted registers, are placed there in `SkeInitSystem`, triggered from a secure call with code `SECURESERVICE_PHASE3_INIT`).

This structure is sent to `ShvlSetRegisterInterceptMasks` which in turn calls `ShvlSetVpRegister` on each of the four values in the input structure to register an intercept. Setting the register values is done by initiating a fast hypercall with a code of `HvCallSetVpRegisters` (`0x51`), sending on four arguments (for anyone interested, all hypercall values are documented here). The last two arguments are of types HV_REGISTER_NAME and HV_REGISTER_VALUE – these types are documented so it's easy to see what registers are being set:



```
NTSTATUS __fastcall ShvlSetRegisterInterceptMasks(_REGISTER_INTERCEPT_INFORMATION *InterceptData)
{
  NTSTATUS status; // eax
  HV_REGISTER_VALUE RegisterValue; // [rsp+20h] [rbp-18h] BYREF

  RegisterValue.Fp.AsUINT128.High64 = 0i64;
  if ( !ShvlpHandleRegisterIntercept && (InterceptData->InterceptControl.AsUINT64 & 0x78007) != 0
    || !ShvlpHandleMsrIntercept && (InterceptData->InterceptControl.AsUINT64 & 0x1EF87FF8) != 0
    || (InterceptData->InterceptControl.AsUINT64 & 0xFFFFFFFE1000000ui64) != 0 )
  {
    return STATUS_INVALID_PARAMETER;
  }
  RegisterValue.Reg64 = InterceptData->Cr0Value;
  status = ShvlSetVpRegister(0xFFFFFFFE, 255, HvX64RegisterCrInterceptCr0Mask, &RegisterValue);
  if ( status >= 0 )
  {
    RegisterValue.Reg64 = InterceptData->Cr4Value;
    status = ShvlSetVpRegister(0xFFFFFFFE, 255, HvX64RegisterCrInterceptCr4Mask, &RegisterValue);
    if ( status >= 0 )
    {
      RegisterValue.Reg64 = InterceptData->Ia32MiscEnableValue;
      status = ShvlSetVpRegister(0xFFFFFFFE, 255, HvX64RegisterCrInterceptIa32MiscEnableMask, &RegisterValue);
      if ( status >= 0 )
      {
        RegisterValue.Reg64 = InterceptData->InterceptControl.AsUINT64;
        return ShvlSetVpRegister(0xFFFFFFFE, 255, HvX64RegisterCrInterceptControl, &RegisterValue);
      }
    }
  }
  return status;
}
```

Looking at the function, we see that it's setting the required values for `CR0`, `CR4` and `IA32_MISC_ENABLE`, and finally setting the mask for intercept control, so from this point all requested registers are intercepted by the hypervisor and any access to them will be forwarded to the SKPG intercept routine.

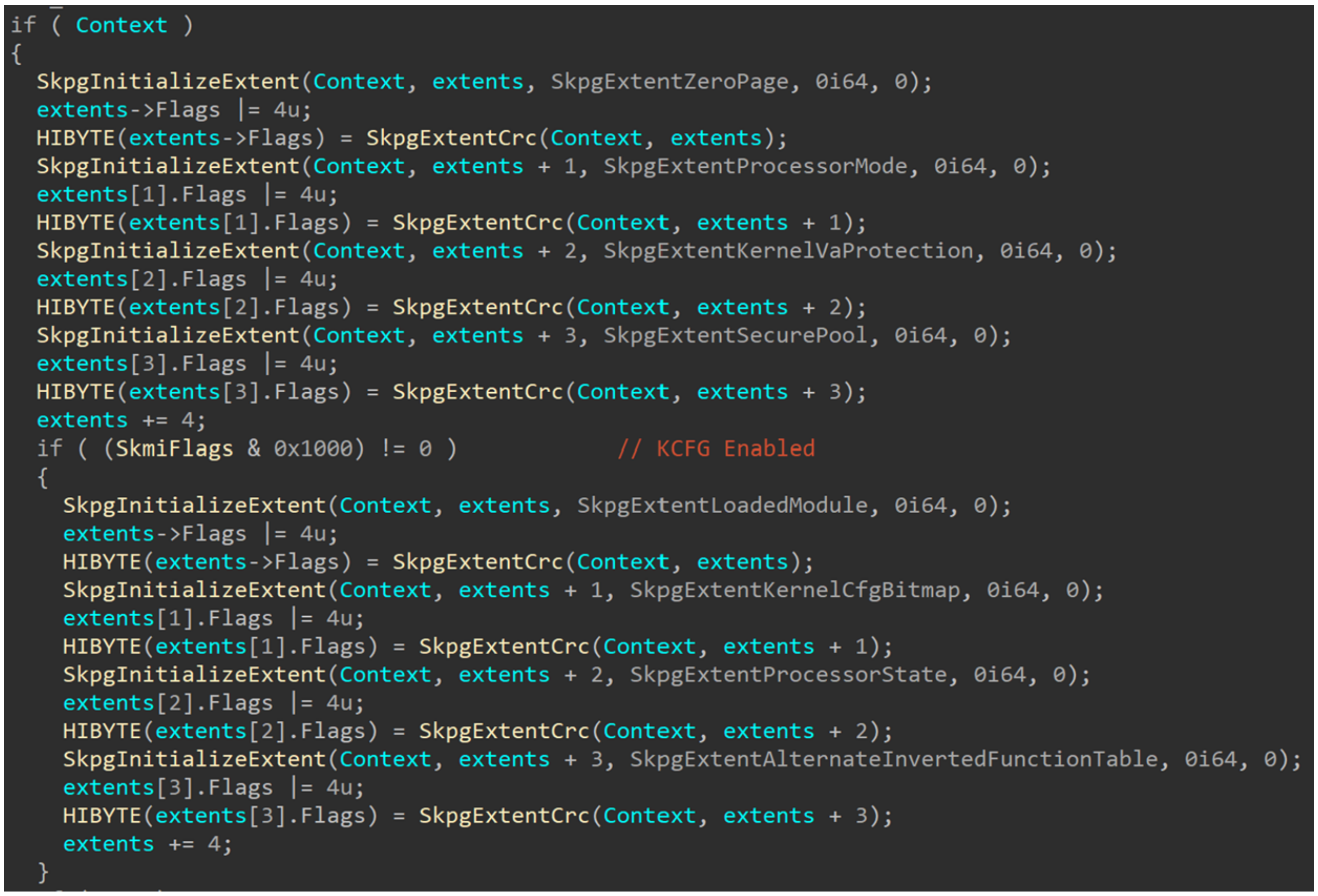
Secure VA Translation Extents

In the previous post I introduced the secure extents – extents indicating `VTL1` memory or data structures to be protected. I also covered memory extents, including the secure memory extents. Here is another kind of secure extents, which are initialized internally in the secure kernel, without using input extents from `VTL0`. They are called Secure VA Translation Extents and are initialized inside `SkpgCreateSecureVaTranslationExtents`. These extents are used to protect Virtual->Physical address translations for different pages or memory regions that are a common target for attack:

- `0x100B`: `SkpgExtentProcessorMode`
- `0x100E`: `SkpgExtentLoadedModule`
- `0x100F`: `SkpgExtentProcessorState`
- `0x1010`: `SkpgExtentKernelCfgBitmap`
- `0x1011`: `SkpgExtentZeroPage`
- `0x1012`: `SkpgExtentAlternateInvertedFunctionTable`
- `0x1015`: `SkpgExtentSecureExtensionTable`
- `0x1017`: `SkpgExtentKernelVAProtection`
- `0x1019`: `SkpgExtentSecurePool`

Though they are called secure extents, the data they protect is mostly `VTL0` data, such as the `VTL0` mapping of the `KCFG` bitmap or the inverted function table. The exact validations done differ between the types: for example, the zero page should never be mapped so a successful virtual->physical address translation of the zero page should not be acceptable, while the kernel `CFG` bitmap should have valid translations but the `VTL0` mapping of those pages should always be read-only.

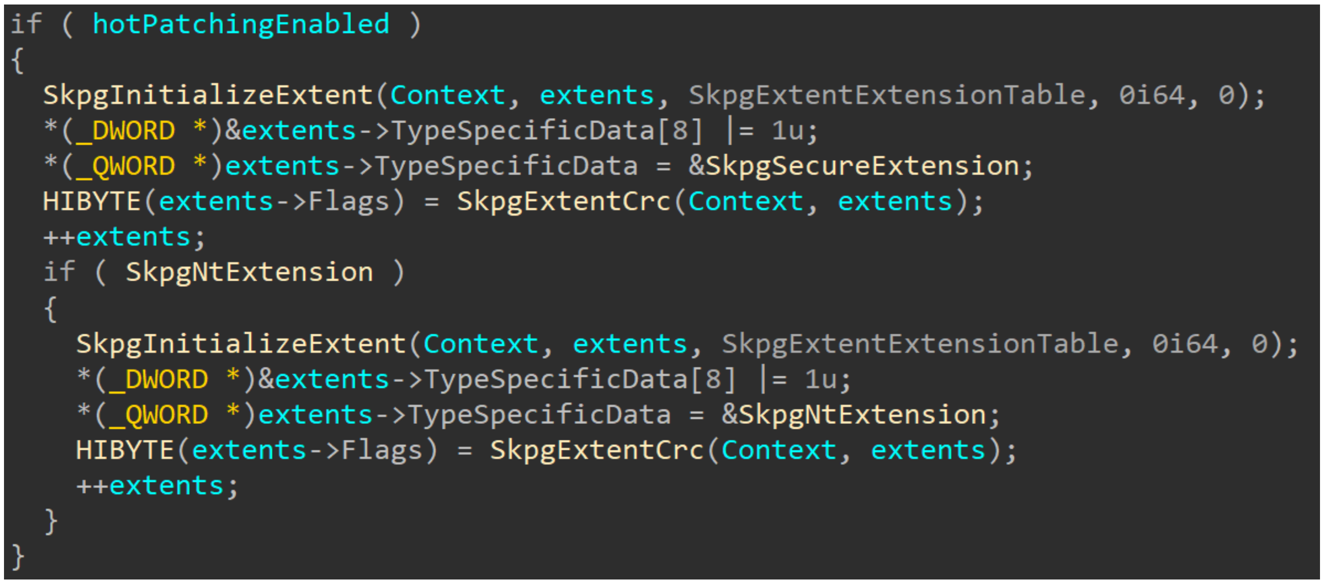
Looking at `SkpgCreateSecureVaTranslationExtents`, we can see that the extents are initialized with no input data or memory ranges:



```
if ( Context )
{
  SkpgInitializeExtent(Context, extents, SkpgExtentZeroPage, 0i64, 0);
  extents->Flags |= 4u;
  HIBYTE(extents->Flags) = SkpgExtentCrc(Context, extents);
  SkpgInitializeExtent(Context, extents + 1, SkpgExtentProcessorMode, 0i64, 0);
  extents[1].Flags |= 4u;
  HIBYTE(extents[1].Flags) = SkpgExtentCrc(Context, extents + 1);
  SkpgInitializeExtent(Context, extents + 2, SkpgExtentKernelVaProtection, 0i64, 0);
  extents[2].Flags |= 4u;
  HIBYTE(extents[2].Flags) = SkpgExtentCrc(Context, extents + 2);
  SkpgInitializeExtent(Context, extents + 3, SkpgExtentSecurePool, 0i64, 0);
  extents[3].Flags |= 4u;
  HIBYTE(extents[3].Flags) = SkpgExtentCrc(Context, extents + 3);
  extents += 4;
  if ( (SkmiFlags & 0x1000) != 0 )            // KCFG Enabled
  {
    SkpgInitializeExtent(Context, extents, SkpgExtentLoadedModule, 0i64, 0);
    extents->Flags |= 4u;
    HIBYTE(extents->Flags) = SkpgExtentCrc(Context, extents);
    SkpgInitializeExtent(Context, extents + 1, SkpgExtentKernelCfgBitmap, 0i64, 0);
    extents[1].Flags |= 4u;
    HIBYTE(extents[1].Flags) = SkpgExtentCrc(Context, extents + 1);
    SkpgInitializeExtent(Context, extents + 2, SkpgExtentProcessorState, 0i64, 0);
    extents[2].Flags |= 4u;
    HIBYTE(extents[2].Flags) = SkpgExtentCrc(Context, extents + 2);
    SkpgInitializeExtent(Context, extents + 3, SkpgExtentAlternateInvertedFunctionTable, 0i64, 0);
    extents[3].Flags |= 4u;
    HIBYTE(extents[3].Flags) = SkpgExtentCrc(Context, extents + 3);
    extents += 4;
  }
```

This is because all of these extents correlate to specific data structures which are all initialized elsewhere so the data doesn't need to be part of the extent itself, so the type is the only part that needs to be set. We can also see that some of these extents are only initialized when `KCFG` is enabled, since without it they are not needed. I will cover the checks done for each of these extents in a later blog post, which will describe SKPG extent verification.

Finally, if HotPatching is enabled, two more extents are added, both with type `SkpgExtentExtensionTable`:



```
if ( hotPatchingEnabled )
{
  SkpgInitializeExtent(Context, extents, SkpgExtentExtensionTable, 0i64, 0);
  *(_DWORD *)&extents->TypeSpecificData[8] |= 1u;
  *(_QWORD *)extents->TypeSpecificData = &SkpgSecureExtension;
  HIBYTE(extents->Flags) = SkpgExtentCrc(Context, extents);
  ++extents;
  if ( SkpgNtExtension )
  {
    SkpgInitializeExtent(Context, extents, SkpgExtentExtensionTable, 0i64, 0);
    *(_DWORD *)&extents->TypeSpecificData[8] |= 1u;
    *(_QWORD *)extents->TypeSpecificData = &SkpgNtExtension;
    HIBYTE(extents->Flags) = SkpgExtentCrc(Context, extents);
    ++extents;
  }
}
```

These extents protect the `SkpgSecureExtension` and `SkpgNtExtension` variables, which keep track of HotPatching data.

Per-Processor Extents

There are two more extents that are processor-specific, since the data they protect exists separately in each processor. However, unlike the MSR and Control Register extents, no intercepts need to be installed and no function needs to be executed on all processors (for now). These extents are also received from the normal kernel and added to the array of extents in the `SKPG_CONTEXT` structure. The data received for each of these two extents includes base address, limit and a processor number, so multiple entries might exist for these extent types, with different processor numbers:

- `0x1004` : `SkpgExtentIdt`
- `0x1005` : `SkpgExtentGdt`

These extents contain the memory range for the `GDT` and `IDT` tables on each processor, so HyperGuard will protect them from malicious modifications.

Unused Extents

Extent types `0x1007` , `0x1008` , `0x1013` and 0x1018 never get initialized anywhere in `SecureKernel.exe` and don't seem to be used anywhere. They may be deprecated or not fully implemented yet.