HyperGuard – Secure Kernel Patch Guard: Part 2 – SKPG Extents

windows-internals.com/hyperguard-secure-kernel-patch-guard-part-2-skpg-extents

By Yarden Shafir

Welcome to Part `2` of the series about Secure Kernel Patch Guard, also known as HyperGuard. This part will start describing the data structure and components of SKPG, and more specifically the way it's activated. If you missed Part `1`, you can find it right here.

Inside HyperGuard Activation

In Part `1` of the series I introduced `HyperGuard` and described its different initialization paths. Whichever path we went through, we end up reaching `SkpgConnect` when the normal kernel finished its initialization. This is when all important data structures in the kernel have already been initialized and can start being monitored and protected by `PatchGuard` and `HyperGuard`.

After a couple of standard input validations, `SkpgConenct` acquires `SkpgConnectionLock` and checks the `SkpgInitialized` global variable to tell if `HyperGuard` has already been initialized. If the variable is set, the function will return `STATUS_ACCESS_DENIED` or `STATUS_SUCCESS`, depending on the information received. In either of those cases, it will do nothing else.

If SKPG has not been initialized yet, `SkpgConnect` will start initializing it. First it calculates and saves multiple random values to be used in several different checks later on. Then it allocates and initializes a context structure, saved in the global `SkpgContext`. Before we move on to other SKPG areas, it's worth spending a bit of time talking about the SKPG context.

SKPG Context

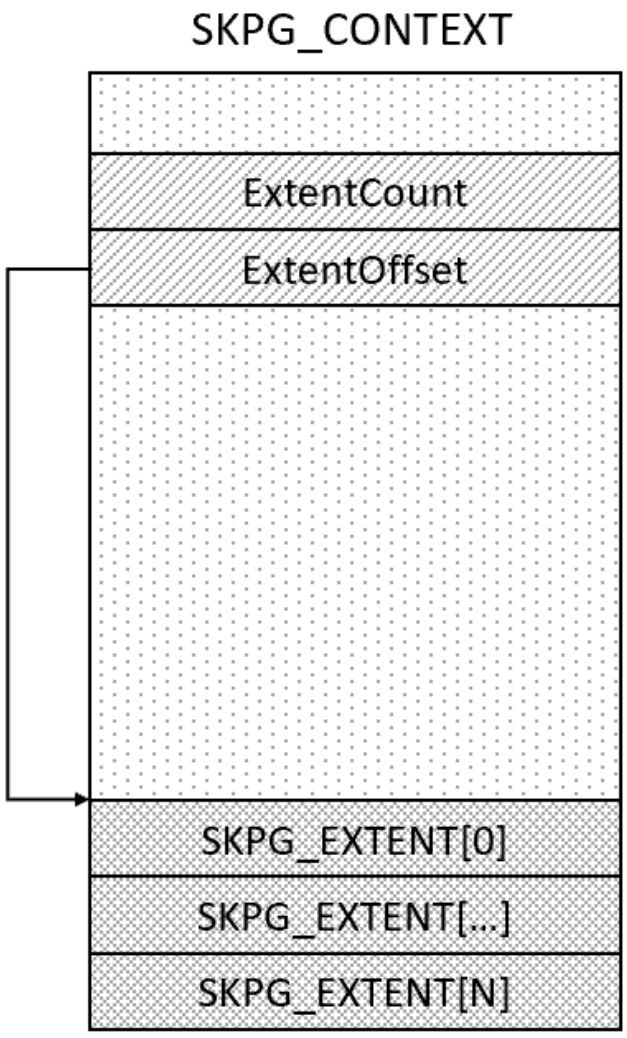
This SKPG context structure is allocated and Initialized in `SkpgConnect` and will be used in all SKPG checks. It contains all the data needed for `HyperGuard` to monitor and protect the system, such as the NT PTE information, encryption algorithms, KCFG ranges, and more, as well as another timer and callback, separate to the ones we saw in the first part of the series. Unfortunately, like the rest of `HyperGuard`, this structure, which I'll call `SKPG_CONTEXT`, is not documented and so we need to do our best to figure out what it contains and how it's used.

First, the context needs to be allocated. This context has a dynamic size that depends on the data received from the normal kernel. Therefore, it is calculated at runtime using the function `SkpgComputeContextSize`. The minimal size of the structure is `0x378` bytes (this number tends to increase every few Windows builds as the context structure gains new fields) and to that will be added a dynamic size, based on the data sent from the normal kernel.

That input data, which is only sent when SKPG is initialized through the `PatchGuard` code paths, is an array of structures named Extents. These extents describe different memory regions, data structures and other system components to be protected by `HyperGuard`. I will cover all of these in more detail later in the post, but a few examples include the `GDT` and `IDT`, data sections in certain protected modules and MSRs with security implications.

After the required size is calculated, the `SKPG_CONTEXT` structure is allocated and some initial fields are set in `SkpgAllocateContext`. A couple of these fields include another secure timer and a related callback, whose functions are set to `SkpgHyperguardTimerRoutine` and `SkpgHyperguardRuntime`. It also sets fields related to PTE addresses and other paging-related properties, since a lot of the `HyperGuard` checks validate correct Virtual->Physical page translations.

Afterwards, `SkpgInitializeContext` is called to finish initializing the context using the extents provided by the normal kernel. This basically means iterating over the input array, using the data to initialize internal extent structures, that I'll call `SKPG_EXTENT`, and sticking them at the end of the `SKPG_CONTEXT` structure, with a field I chose to call `ExtentOffset` pointing to the beginning of the extent array (notice that none of these structures are documented, so all structure and field names are made up):



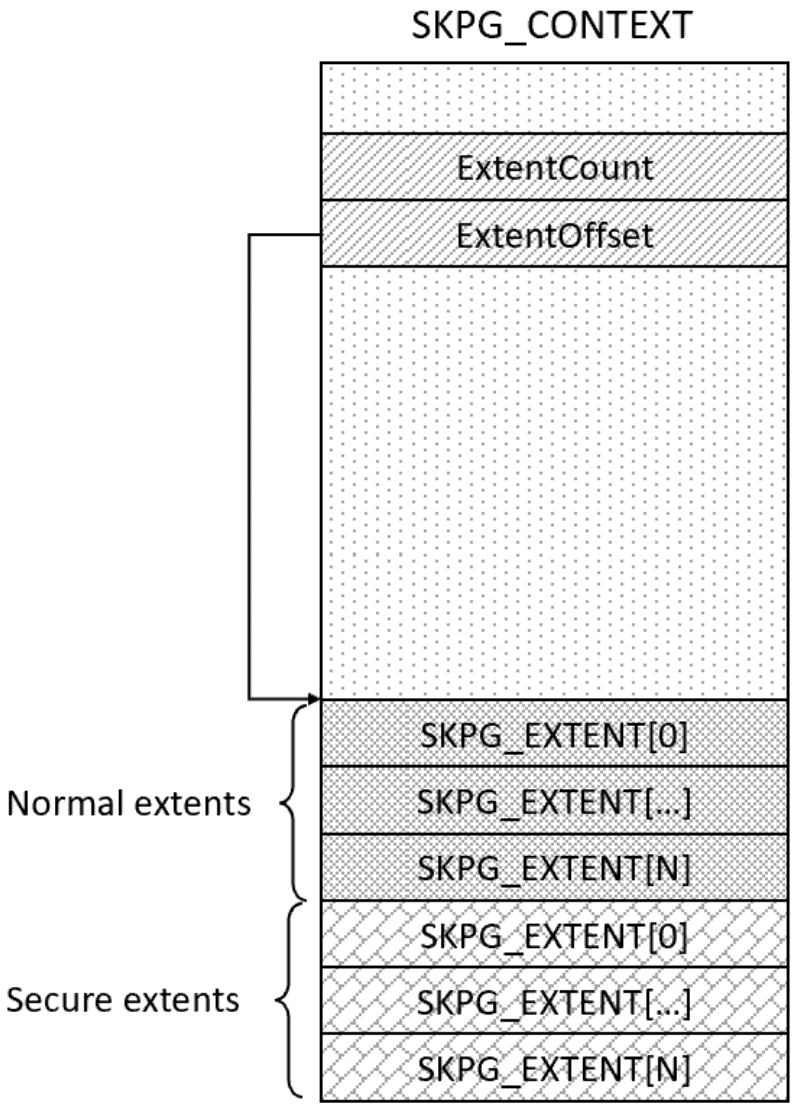
SKPG Extents

There are many different types of extents, and each `SKPG_EXTENT` structure has a `Type` field indicating its type. Each extent also has a hash, used in some cases to validate that no changes were done to the monitored memory region. Then there are fields for the base address of the monitored memory and the number of bytes, and finally a union that contains data unique to each extent type. For reference, here is the reverse engineered `SKPG_EXTENT` structure:

```
typedef struct _SKPG_EXTENT
{
    USHORT Type;
    USHORT Flags;
    ULONG Size;
    PVOID Base;
```

```
    ULONG64 Hash;
    UCHAR TypeSpecificData[0x18];
} SKPG_EXTENT, *PSKPG_EXTENT;
```

I mentioned that the input extents used by `HyperGuard` were provided by the `PatchGuard` initializer function in the normal kernel. But SKPG initializes another kind of extents as well – secure extents. To initialize those, `SkpgInitializeContext` calls into `SkpgCreateSecureKernelExtents`, providing the `SKPG_CONTEXT` structure and the address where the current extent array ends – so the secure extents can be placed there. Secure extents use the same `SKPG_EXTENT` structure as regular extents and protect data in the secure kernel, such as modules loaded into the secure kernel and secure kernel memory ranges.



Extent Types

Like I mentioned, there are many different types of extents, each used by `HyperGuard` to protect a different part of the system. However, we can split them into a few groups that share similar traits and are handled in a similar way. For clarity and to separate normal extents from secure extents, I will use the naming convention `SkpgExtent` for normal extent types and `SkpgExtentSecure` for secure extent types.

The first extent that I'd like to cover is a pretty simple one that always gets sent to `SkpgInitializeContext` regardless of other input:

Initialization Extent

There is one extent that doesn't belong in any of the groups since it is not involved in any of the `HyperGuard` validations. This is extent `0x1000` : `SkpgExtentInit` – this extent is not copied to the array in the context structure. Instead, this extent type is created by `SkpgConnect` and sent into `SkpgInitializeContext` to set some fields in the context structure itself that were previously unpopulated. These fields have additional hashes and information related to hotpatching, such as whether it is enabled and the addresses of the retpoline code pages. It also sets some flags in the context structure to reflect some configuration options in the machine.

Memory and Module Extents

This group includes the following extent types:

- `0x1001` : `SkpgExtentMemory`
- `0x1002` : `SkpgExtentImagePage`
- `0x1009` : `SkpgExtentUnknownMemoryType`
- `0x100A` : `SkpgExtentOverlayMemory`
- `0x100D` : `SkpgExtentSecureMemory`
- `0x1014` : `SkpgExtentPartialMemory`
- `0x1016` : `SkpgExtentSecureModule`

The thing all these extent types have in common is that they all indicate some memory range to be protected by `HyperGuard` . Most of these contain memory ranges in the normal kernel, however `SkpgExtentSecureMemory` and `SkpgExtentSecureModule` have `VTL1` memory ranges and modules. Still, all these extent types are handled in a similar way regardless of the memory type or `VTL` so I grouped them together.

When normal memory extents are being added to the SKPG Context, all normal kernel address ranges get validated to ensure that the pages have a valid mapping for SKPG protection. For a normal kernel page to be valid for SKPG protection, the page can't be writable. SKPG will monitor all requested pages for changes, so a writable page, whose contents can change at any time, is not a valid "candidate" for this kind of protection.

Therefore, SKPG can only monitor pages whose protection is either “read” or “execute”. Obviously, only valid pages (as indicated by the Valid bit in the PTE) can be protected. There are slight differences to some of the memory extents when HVCI is enabled as SKPG can’t handle certain page types in those conditions.

Once mapped and verified, each memory page that should be protected gets hashed, and the hash gets saved into the `SKPG_EXTENT` structure where it will be used in future `HyperGuard` checks to validate that the page wasn’t modified.

Some memory extents describe a generic memory range, and some, like `SkpgExtentImagePage`, describe a specific memory type that needs to be treated slightly differently. This extent type mentions a specific image in the normal kernel, but `HyperGuard` should not be protecting the whole image, only a part of it. So the input extent has the image base, the page offset inside the image where the protection should start and the requested size. Here too the memory region to be protected will be hashed and the hash will be saved into the `SKPG_EXTENT` to be used in future validations.

But the `SKPG_EXTENT` structures that get written into the SKPG Context normally only describe a single memory page while the system might want to protect a much larger area in an image. It is simply easier for `HyperGuard` to handle memory validations one page at a time, to make for more predictable processing time and avoid taking up too much time while hashing large memory ranges, for example. So, when receiving an input extent where the requested size is larger than a page (`0x1000` bytes), `SkpgInitializeContext` iterates over all the pages in the requested range and creates a new `SKPG_EXTENT` for each of them. Only the first extent, describing the first page in the range, receives the type `SkpgExtentImage`. All the other ones that describe the following pages receive a different type, `0x1014`, which I chose to call `SkpgExtentPartialMemory`, and the original extent type is placed in the first `2` bytes in the type-specific data inside the `SKPG_EXTENT` structure.

Every extent in the array can be marked by different flags. One of these is the `Protected` flag, which can only be applied to normal kernel extents, meaning that the specified address range should be protected from changes by SKPG. In this case, `SkpgInitializeContext` will call `SkmmPinNormalKernelAddressRange` on the requested address range to pin in and prevent it from being freed by `VTL0` code:

```
      protectRange = SkpgProtectedPagesExist((unsigned __int64)ADJ(inputExtent)->Base, Size);
      flags = 8;
      if ( !protectRange )
        flags = 0;
      newExtent->Flags = flags | newExtent->Flags & 0xFFF7;
    }
    *(_OWORD *)newExtent->TypeSpecificData = *(_OWORD *)&ADJ(inputExtent)->VariableData[4];
LABEL_53:
    if ( (newExtent->Flags & 8) != 0 )          // Protected flag
    {
      status_1 = SkmmPinNormalKernelAddressRange((unsigned __int64)ADJ(inputExtent)->Base, ADJ(inputExtent)->Size);
      v12 = 0i64;
      if ( status_1 < 0 )
      {
        newExtent->Flags &= ~8u;
        if ( status_1 != (unsigned int)STATUS_NOT_FOUND )
        {
LABEL_60:
          status = STATUS_INVALID_PARAMETER;
          goto LABEL_73;
        }
      }
    }
```

The secure memory extents essentially behave very similar to the normal memory extent, with the main differences being that they are initialized by the secure kernel itself and the details of what they are protecting.

Extents of type `SkpgExtentSecureModule` are generates to monitor all images loaded into the secure kernel space. This is done by iterating the `SkLoadedModuleList` global list, which, like the normal kernel's `PsLoadedModuleList`, is a linked list of `KLDR_DATA_TABLE_ENTRY` structures representing all loaded modules. For each one of those modules, `SkpgCreateSecureModuleExtents` is called to generate the extents.

To do so, `SkpgCreateSecureModuleExtents` receives a `KLDR_DATA_TABLE_ENTRY` for one loaded DLL at a time, validates that it exists in `PsInvertedFunctionTable` (a table containing basic information for all loaded DLLs, mostly used for quick search for exception handlers) and then enumerates all the sections in the module. Most sections in a secure module are monitored using an `SKPG_EXTENT` but are not protected from modifications. Only one section is being protected, the `TABLERO` section:

```
sectionIsTableRO = *(_DWORD *)imageSectionHeader->Name == 'LBAT' && *(_DWORD *)&imageSectionHeader->Name[4] == 'ORE';
```

```
if ( sectionIsTableRO )
{
  SkmmProtectKernelImageSubsection((ULONG_PTR)dllBase, (ULONG_PTR)imageSectionHeader);
  v16 = Context;
}
sectionIsExecutable = (imageSectionHeader->Characteristics >> 29) & 1;// IMAGE_SCN_MEM_EXECUTE
if ( (_DWORD)numberOfExtentsForSection )
{
  pbInput = sectionBase;
  do
  {
    SkpgInitializeSecureMemoryExtent(v16, v5, 0x1016, v6, sectionIsExecutable, pbInput, size);
    v16 = Context;
    ++v5;
    ++v6;
    pbInput += size;
    LODWORD(numberOfExtentsForSection) = numberOfExtentsForSection - 1;
  }
  while ( (_DWORD)numberOfExtentsForSection );
```

The `TABLERO` section is a data section that exists in only a handful of binaries. In the normal kernel it exists in Win32k.sys, where it contains the win32k system service table. In the secure kernel a `TABLERO` section exists in securekernel.exe, where it contains global variables such as `SkiSecureServiceTable`, `SkiSecureArgumentTable`, `SkpgContext`, `SkmiNtPteBase`, and others:

```
0000001400FD000 ; Segment type: Pure data
0000001400FD000 ; Segment permissions: Read/Write
0000001400FD000 TABLERO         segment para public 'DATA' use64
0000001400FD000                 assume cs:TABLERO
0000001400FD000                 ;org 1400FD000h
0000001400FD000 SkiSecureServiceTable dq offset IumEmitSmc
0000001400FD000                                         ; DATA XREF: SkiSystemStartup+102↑o
0000001400FD000                                         ; SkiCompactSecureServiceTable+6↑o ...
0000001400FD008                 dq offset IumEmitSmc
0000001400FD010                 dq offset IumCreateSecureDevice
0000001400FD018                 dq offset IumCreateSecureSection
0000001400FD020                 dq offset IumCreateSecureSectionSpecifyPages
0000001400FD028                 dq offset IumCrypto
0000001400FD030                 dq offset IumDmaMapMemory
0000001400FD038                 dq offset IumEmitSmc
0000001400FD040                 dq offset IumFlushSecureSectionBuffers
0000001400FD048                 dq offset IumGetDmaEnabler
0000001400FD050                 dq offset IumGetExposedSecureSection
0000001400FD058                 dq offset IumGetIdk
0000001400FD060                 dq offset IumMapSecureIo
0000001400FD068                 dq offset IumEmitSmc
0000001400FD070                 dq offset IumOpenSecureSection
0000001400FD078                 dq offset IumPostMailbox
0000001400FD080                 dq offset IumProtectSecureIo
0000001400FD088                 dq offset IumQuerySecureDeviceInformation
0000001400FD090                 dq offset IumSecureStorageGet
0000001400FD098                 dq offset IumSecureStoragePut
0000001400FD0A0                 dq offset IumSetDmaTargetProperties
0000001400FD0A8                 dq offset IumSetPolicyExtension
0000001400FD0B0                 dq offset IumUnmapSecureIo
0000001400FD0B8                 dq offset IumUpdateSecureDeviceState
0000001400FD0C0 SkiSecureServiceLimit dd 18h            ; DATA XREF: SkiCompactSecureServiceTable↑r
0000001400FD0C0                                         ; KiSystemCall64+125↑r
0000001400FD0C4 SkiSecureArgumentTable db  20h          ; DATA XREF: SkiCompactSecureServiceTable+D↑o
```

When `SkpgCreateSecureModuleExtents` encounters a `TABLERO` section, it calls `SkmmProtectKernelImageSubsection` to change the PTE for the section pages from the default read-write to read only.

Then for each section, regardless of its type, an extent with type `SkpgExtentSecureModule` is created. Each memory region gets hashed a flag in the extent marks if the section is executable. The number of extents generated per section can vary: If HotPatching is enabled on the machine a separate extent will be generated for every page in the protected image ranges. Otherwise, every protected section generates one extent that might cover multiple pages, all of them with type `SkpgExtentSecureModule` :

```
size = sizeOfRawData;
if ( hotPatchEnabled )
{
  size = 0x1000;
  numberOfExtentsForSection = ((((_DWORD)dllBase + imageSectionHeader->VirtualAddress) & 0xFFF)
                              + (unsigned __int64)sizeOfRawData
                              + 0xFFF) >> 12;
}
else
{
  LODWORD(numberOfExtentsForSection) = 1;
}
```

If HotPatching is enabled, one last secure module extent gets created for each secure module. The variable `SkmiHotPatchAddressReservePages` will indicate how many pages are reserved for HotPatch use at the end of the module, and an extent gets created for each of those pages. Similar to the way described earlier for normal kernel module extents, each extent describes a single page, the extent type is `SkpgExtentPartialMemory` and the type `SkpgExtentSecureModule` is placed in one of the type-specific fields of the extent.

Another secure extent type is `SkpgExtentSecureMemory` . This is a generic extent type used to indicate any memory range in the secure kernel. However, for now it is only used to monitor the `GDT` pointed to by the secure kernel processor block – the `SKPRCB` . This is an internal structure that is similar in its purpose to the normal kernel's `KPRCB` (and similarly, an array of them exists in `SkeProcessorBlock`). There will be one extent of this type for each processor in the system. Additionally, the function sets a bit in the Type field of each `KGDTENTRY64` structure to indicate that this entry has been accessed and prevent it from being modified later on – but the entry for the `TSS` at offset `0x40` gets skipped:

```
skprcb = SkeProcessorBlock[number];
if ( size + 0x30 < size )
{
    break;
}
++count;
size += 0x30i64;
if ( Context )
{
    Gdt = (_KGDTENTRY64 *)skprcb->Gdt;
    for ( i = 0i64; i < 7; ++i )
    {
        gdtEntry = GdtEntriesOffsets[i];
        *(_DWORD *)((char *)&Gdt->Bits + gdtEntry) |= 0x100u;
    }
    SkpgInitializeSecureMemoryExtent(
        Context,
        v9++,
        SkpgExtentSecureMemory,
        count - 1,
        1u,
        skprcb->Gdt,
        0x50u);
}
```

This pretty much covers the initialization and uses of the memory extents. But this is just the first group of extents, and there are many others that monitor various different parts of the system. In the next post I'll talk about more of these other extent types, which interact with system components like MSRs, control registers, the `KCFG` bitmap and more!