Inside Windows Defender System Guard Runtime Monitor

blog.syscall.party/2022/08/02/inside-windows-defender-system-guard-runtime-monitor.html

$~ lloydlabs August 2, 2022

Aug 2, 2022

What is System Guard Runtime Monitor? (SGRM)

System Guard Runtime Monitor (SGRM) is a component of Windows Defender (WD), that was introduced in the Windows 10 1709 update and has been present since as a key component to ensure system integrity.

Another name for this component is Octagon, which is assumed to be an internal project name for Microsoft, where System Guard Runtime Monitor is used as the public name for marketing Windows Defender. For SGRM to work, a device must have Virtual Secure Mode enabled on their system, as the protection makes use of Virtual Trust Levels to minimize the attack surface on the core attestation Lua engine. Microsoft, in **this** 2018 blog, describe SGRM as `"If important security features should fail, users should be aware. Windows Defender System Guard runtime attestation, a new Windows platform security technology, fills this need"`. Simply put, SGRM is an anti-tampering mechanism for your modern Windows device.

In this post, I'll go into the details behind how SGRM works, the Lua component, integrity checks performed, the RPC service, and more from a brief reverse engineering standpoint of this WD component.

Components

We'll quickly take a look at some of the components of SGRM, to provide some context before going through the details of each individual component.

Component	Usage
SgrmBroker.exe	Provides a client API, exposing assists to the SGRM runtime when doing assertions.
Sgrm.sys	The agent driver, exposes functionality for use within the assertion assists wrappers used by SgrmBroker.
SgrmEnclave.dll	Lua assertion engine, also called the enclave controller shim, contains the Lua runtime, *SgrmEnclave_secure.dll* runs in VTL-1, or another mode of operation. Talks to SgrmBroker.exe via the API.
SgrmLpac.exe	A local RPC service, which exposes a method to send an HTTP POST request to a specified endpoint

SgrmAgent.sys

Driver Boot

The agent driver, `%WINDIR%\System32\drivers\SgrmAgent.sys`, provides kernel-level assists to the SgrmBroker assist engine, which runs in user-mode.

Upon driver boot new secure device is created under `\\Device\\MSSGRMAGENTSYS`, with a symbolic link pointing to this device under `\\??\\MSSGRMAGENTSYS`. An Event Tracing (Etw) provider is registered under `Microsoft.Windows.Oct.Driver`. The provider receives violations for checks the driver implements are sent to throughout the execution.

After this has completed, the driver requests a kernel extension host from ntoskrnl, using the (mostly undocumented) API `ExRegisterHost`, under `PspOctExtensionHost`, from `PspOctExtensionInterface`. The registering of this host provides straight up, direct kernel access to certain functions, almost as if the agent code was running within ntoskrnl itself. A table of extension functions is initialized with a pool tag of `0x2000D`:

```
.data:00000001C00088A8 g_ExtensionFunctions dq 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0; 0
.data:00000001C00088A8                                         ; DATA XREF:
OctpInitialize:loc_1C000B25D↓o
.data:00000001C00088A8                                         ; OctpInitialize+26E↓w ...
```

Initialization

To initialize the driver, an IOCTL `0x9C402480` must be sent to the driver. This then calls the internal function `OctpHandleInitRequest`.

The driver has several checks in place to ensure the integrity of the user-mode caller:

- The device is bound to the SgrmBroker (OctBroker) service SID upon creation with `WdmlibIoCreateDeviceSecure` `"D:P(A;;GRGWGX;;;S-1-5-80-3706850399-3459138796-2835936764-562029542-397710147)`
- Within the dispatch routine, the driver can only be initialized once, resulting only in a single handle at a time effectively being open
- Only Windows TCB signed processes (not to be confused with Thread Control Block) can open a `HANDLE` (Windows Defender runs as a Protected Process Light (PPL) under the anti-malware context, the trust level would not suffice)

For example, below, if the driver has already been initialized in `OctpHandleInitRequest`, the driver will write an event to the abovementioned Etw provider and deny access.

```
// Global g_initialized flag check ≠ 0
if ( _InterlockedCompareExchange(&g_initialized, 1, 0) == 1 )
{
    EtwWriteTransfer(1, g_SgrmDescriptor, {"OctpHandleInitRequest", "SGRM device was already
initialized!"}, ...);
}
```

Agent Mailbox

As previously mentioned, the driver exposes functionality to `SgrmBroker`. The IOCTL `0x9C402484` is used to dispatch to the specified function within the driver mailbox.

```
struct _SGRM_MAILBOX_REQ {
    UINT8 u8Index;
    DWORD dwArgNumb;
    [function arguments]
} *PSGRM_MAILBOX_REQ, SGRM_MAILBOX_REQ;
```

The list of driver assist functions can be observed in the table below.

The specific mailbox value is passed in a struct, to SgrmAgent.sys's `OctpMailboxDispatcher`. Below, is an excerpt of the function targets that can be accessed by the user-mode agent for utility:

Idx	Mailbox Target	Description
0x1	OctpHandleMapVirtualAddress	Map a specified virtual address (including kernel mode)
0x2	OctpHandleUnmapVirtualAddress	Unmap a specified virtual address
0x13	OctpHandleGetThreadContext	Get the context of specified thread
0x15	OctpHandleGetMsrValue	Read the value of a specified MSR
0x12	OctpHandleGetMemoryRegionInfo	Get the information of a specified memory region
0xD	OctpHandleFreezeThread	Freeze a specified thread
0xE	OctpHandleThawThread	Similar to freeze thread
0xF	OctpHandleCopyMemory	Copy memory (including physical), destination, source, and size.
0x10	OctpHandleMapFile	Map a specified file
0x11	OctpHandleUnmapFile	Unmap a specified file
0xC	OctpHandleGetStructureOffsetSize	Get the offset of a specified structure, e.g. `EPROCESS`

Along with providing these utilities, the agent provides access to these objects to the broker through the mailbox for runtime inspection via the engine on demand:

- `EPROCESS`
- `ETHREAD`
- `DRIVER_OBJECT`
- `TOKEN`

SgrmBroker.exe

Service Start

`SgrmBroker.exe` is one of the user-mode components of the System Guard Runtime Monitor, and runs as a service named `OctBroker` (or, `SgrmBroker`). The service is started as a delayed service for performance reasons, and has a trigger to spawn the client API service upon boot. The process runs as a signed Windows Trusted Computing Base (TCB) PPL.

When the service starts, the following is applied to the process, for performance and security:

- Set the current thread priority to `THREAD_MODE_BACKGROUND_BEGIN`
- Set the `PROCESS_INFORMATION_CLASS` of the current process to `ProcessEnableLogging`
- Enable ASLR using `ProcessASLRPolicy`
- Enable handle checking using `ProcessStrictHandleCheckPolicy`
- Set the heap to terminate upon corruption using `HeapEnableTerminationOnCorruption`

The broker then does some core initialization for general functionality, such as setting up an Etw handler. The LAPC component (`SgrmLpac.exe`) is programmatically spawned within the initialization function, all within the `LpacHost.exe` class. A job is created for it through `CreateJobObjectA` , and the handle of the `LpacHost.exe` process is bound to it. This class provides the functionality to spawn, terminate, and use the HTTP POST RPC endpoint. After the RPC server has spawned, if the process is terminated, `LpacHost::LpacTerminatedCallback` is called which closes all of the RPC connections. The details of this component are examined in the *SgrmLpac* section of this blog post.

Once the core initialization has completed, a path to `%WINDIR%\System32\Sgrm` is constructed. To ensure the integrity of this directory, an event is created `DirectoryWatcher` class to watch for tampering efforts that may take place to the files in this directory. A thread is then spawned in the background to watch the directory under `DirectoryWatcher::StartWatch` . If one of the watched files is altered, an Etw write event is triggered. The files in this directory are detailed below:

File	Usage
SgrmAssertions.bin	Compiled Lua script, with Microsoft-defined header, which contains the assertions
SgrmAssertions.cat	Certificate used to verify the assertions that the engine makes, to ensure they are legitimate

Then, the RPC client API is initialized as a local service (`ncalrpc`) within `OctagonRpcInitialize` .

initialization

The broker attempts to open a handle to the agent driver - `\\\\.\\MSSGRMAGENTSYS` . If it is not successful, possibly indicating a malicious handle to the driver, an Etw event is triggered. If this is successful, the initialization IOCTL is sent (`0x9C402480`) via to the fast I/O control port in the driver. If the driver has already been initialized, as discussed in the previous section, this will fail.

The number of bytes returned from the `DeviceIoControl` call must be `>= 0x11D` , if not, a `ValidateResponseSize` Etw event is triggered. A structure is populated with information from the `OctpGetNtosInfo` within the driver. The `OctpGetNtosInfo` queries the `SystemModuleInformationEx` class by calling `ZwQuerySystemInformation` to populate the undocumented `RTL_PROCESS_MODULE_INFORMATION_EX` structure, only certain applicable members are used in the resultant buffer.

```
// Credit: ProcessHacker
typedef struct _RTL_PROCESS_MODULE_INFORMATION_EX
{
    USHORT NextOffset;
    struct _RTL_PROCESS_MODULE_INFORMATION
    {
        HANDLE Section;
        PVOID MappedBase;
        PVOID ImageBase;
        ULONG ImageSize;
        ULONG Flags;
        USHORT LoadOrderIndex;
        USHORT InitOrderIndex;
        USHORT LoadCount;
        USHORT OffsetToFileName;
        UCHAR FullPathName[256];
    } BaseInfo;
    ULONG ImageChecksum;
    ULONG TimeDateStamp;
    PVOID DefaultBase;
} RTL_PROCESS_MODULE_INFORMATION_EX, *PRTL_PROCESS_MODULE_INFORMATION_EX;
```

Virtual Secure Mode Enclave Host Controller

The broker has a helper class named `EnclaveControllerVsm` in which `SgrmEnclave_secure.dll` resides. `SgrmEnclave_secure.dll` is the assertion engine, with the Lua engine embedded. Upon initialization, `EnclaveControllerVsm` checks if the enclave `ENCLAVE_TYPE_VBS` is supported by the processor. If supported, a enclave is then created, and the assertion engine is loaded into it. The details of the assertion engine can be found in the `SgrmEnclave_secure.dll` section of this blog post.

The secure enclave is then initialized, and a pointer to `EngHostAttest` is resolved from the global array of function names (`g_engHostFunctionMappings`), and called.

The shim DLL (SgrmEnclave.dll) is also initialized within `EnclaveControllerShim` , `EnclaveControllerShim` . `EngHostAttest` is then resolved and called from the same function table. This does not run in an VSM enclave.

Modes Of Operationi

VTL-1 is not always avaliable on a machine, for example an older system may not support Virtual Secure Mode or have a TPM chip. Below are the modes in which the enclave can run in when verifying the result of an attestation.

#	Mode	Description
1	Virtualization Based Security (VBS) + Dynamic Root of Trust for Measurement (DTRM)	Keys used to verify attestation placed in the TPM's PCR registers, allocated within $pcr17-23.
2	VBS	Keys used to verify attestation use host keys
3	None	No attestation, no keys used - not supported by the host

Assists

The broker provides a number of assists which the assertion engine requires, as it is simply a Lua script. The assists are needed for the script to interact with the system to make the assertions based of certain system information. These assists are stored and initialized at runtime, and passed to the assertion engine upon startup. They are stored in a table named `g_BrokerAssistCallbackTbl` of type `BROKER_CALLBACK_TABLE`, and use the following format.

```
struct _BROKER_CALLBACK_TABLE g_BrokerAssistCallbackTbl
?g_BrokerAssistCallbackTbl@@3U_BROKER_CALLBACK_TABLE@@A dq offset ?
dq offset ?Broker_CancelTimer@Assists@@SAPEAXPEAX@Z ; Assists::Broker_CancelTimer(void *)
dq offset ?Broker_CloseHandle@Assists@@SAPEAXPEAX@Z ; Assists::Broker_CloseHandle(void *)
dq offset ?Broker_CreateEvent@Assists@@SAPEAXPEAX@Z ; Assists::Broker_CreateEvent(void *)
```

Some assists are simply user-mode calls, such as registry reads, and some require using the `SgrmAgent.sys` driver for extended functionality.

Assists which require the usage of the agent driver are prefixed with `Assists::Agent_`, an excerpt of these mapped to the internal agent functions are below.

Broker Assist Function	Driver Mailbox Function
`Assists::Agent_ReadMsr`	`OctpHandleGetMsrValue`
`Assists::Agent_ThawThread`	`OctpHandleThawThread`
`Assists::Agent_UnmapFile`	`OctpHandleUnmapFile`
`Assists::Agent_MapVirtualAddress`	`OctpHandleMapVirtualAddress`

As mentioned, some of the assists that can be done in user-mode, and don't require use of the driver. These are appended with `Assists::Broker_`. For example, the assist for a `Sleep` is as trivial as it sounds:

```
void Assists::Broker_Sleep(DWORD *dwSleepMs)
{
  Sleep(*dwSleepMs);
  return 0;
}
```

SgrmEnclave_secure.dll

The engine relies on the broker to load the Lua script from disk, meaning the DLL which is within the Secure Enclave does perform any file I/O for security reasons - and is instead wrapped through an assist as previously covered. All of the assits within the function are called through a wrapper class, subsequently named `AssistWrapper`.

The script which is loaded must have a "metadata" header with a specific format. Luckily for our reverse engineering efforts, the source code contains debug strings with the original struct field names. Below is the structure, along with the constraints that must be passed.

```
typedef struct _LUA_FILE_HEADER {
    BYTE TLVEmpty[6]; // [1, 0, 0, 0, 0]
    UINT16 Version; // header->Version == LUA_FILE_HEADER_FORMAT_VERSION
    UINT32 SVN;
    UINT16 MajorScriptVersion; // header->MajorScriptVersion > MAJOR_SCRIPT_VERSION
    UINT16 MinorScriptVersion; // header->MinorScriptVersion > MINOR_SCRIPT_VERSION
    UINT16 MajorEngineVersion; // header->MajorEngineVersion == MAJOR_ENGINE_VERSION
    UINT16 MinorEngineVersion; // header->MinorEngineVersion == MINOR_ENGINE_VERSION
} *PLUA_FILE_HEADER, LUA_FILE_HEADER;
```

The size of the Lua script which is loaded is also checked, it must be greater than the size of the structure. If any of these checks fail, a `EnforceLuaScriptMetadata` Etw event is triggered.

Followed by this header is compiled Lua code.

```
typedef struct _LUA_FILE {
    LUA_FILE_HEADER lfhMetadata;
    BYTE bCompiled[];
} *LUA_FILE, LUA_FILE;
```

The Lua engine lives inside the `SgrmEnclave_secure.dll` DLL and is initialized within the `EngInitialize` function. A global lock is issued to ensure the engine doesn't run twice, and a new heap is created for use by the runtime. For unsafe calls inside the engine. e.g. raw function pointers to `NATIVE_*` functions, a special wrapper function `EngpDispatchLuaPcallUnsafe` is used.

The state of the memory region used by the Lua runtime within the secure enclave is saved, any calls must pass a `MemoryUtil::IsBufferOutsideEngine` check - which verifies the memory is within the bounds of `MemoryUtil::EngineStart` and `MemoryUtil::EngineEnd`. This utility is used to ensure mapped files, etc, aren't living outside the enclave region.

Lua Assertions

This is basically the core of SGRM, the assertion engine - where all of the magic is done. All of the assertions are within Lua, and contained within the compiled script which format is described in the previous section. When the script needs to use functionality, the functions are prefixed with `NATIVE_*`, for example, `NATIVE_GetProcessById` - which calls into the function within the assist wrapper in `SgrmEnclave_secure.dll`.

The fully decompiled script can be found in my GitHub repository for this research here. For example, below we can see the reconstructed script reading the MSR `C0010015` for AMD processors.

```
if not msrTable then
    msrTable = {}
    msrTable.CPUVendorId = "AuthenticAMD"
    msrTable.CPUFamily = L14_2
    msrTable.Msr_C0010015 = L18_2
    msrTable = L1_2.rn_b151
    msrTable(L18_2, L19_2, L20_2)
end
```

There are several global variables initialized within the script, some from assists when bootstrapping, such as `GLOBAL_KERNEL_IMAGE_SIZE`:

`GLOBAL_FRAMEWORK_ACCOUNTING`	`GLOBAL_PROFILEENABLED`	`GLOBAL_ASSERTION_TABLE`
`GLOBAL_BOOTID`	`GLOBAL_INITIALIZATION_FINISHED`	`GLOBAL_PAGESIZE`
`GLOBAL_KERNEL_IMAGE_BASE`	`GLOBAL_KERNEL_IMAGE_SIZE`	`GLOBAL_SCRIPT_ID`

The list of assertions can be read within the script; however this list is incomplete as the decompiled code is hard to read. The following list of drivers are checked to *not* be loaded, likely the top malicious drivers in their opinion as there are *many* more on their driver blocklist.

Driver	Description
`\Device\mimidrv`	Mimikatz credential dumping utility driver
`\Device\HackSysExtremeVulnerableDriver`	An intentionally vulnerable open-source driver developed by HackSys
`\Device\Htsysm72FB`	A Capcom driver, which was commonly used by adversaries to achieve code execution in kernel mode

The tampering of critical system processes is also tracked within the Lua script, the following tokens are verified against these processes:

Process	Token
`SgrmBroker.exe`	PsProtectedSignerWinTcb
`MsMpEng.exe`	PsProtectedSignerAntimalware
`crss.exe`	PsProtectedSignerWinTcb
`MsMpEngCP.exe`	PsProtectedSignerAntimalware
`MsSense.exe`	PsProtectedSignerWindows

For firmware integrity, the MSRs of these CPU manufacturers are checked:

Manufacturer	MSR
AMD	`C0010015`
Intel	`C80`

The integrity of several critical, and also Microsoft anti-malware, system drivers are checked:

Driver	Description
`\Driver\mssecflt`	Microsoft Security Events Component Minifilter
`\Driver\WdFilter`	Microsoft Defender Antivirus Mini-Filter Driver
`\Driver\SgrmAgent`	System Guard Runtime Monitor Agent Driver, our favourite!

By using the `DRIVER_OBJECT` returned by assist functions, the following routines within the driver are checked. A state at first run is saved, and any changes trigger an assertion failure:

- `DriverObject->FastIoDispatch`
- `DriverObject->MajorFunction`
- `DriverObject->DriverStartIo`
- `DriverObject->DriverUnload`
- `DriverObject->DriverStart`
- `DriverObject->DriverSize`
- `DriverObject->DriverName`
    `Length` and `Buffer` are both checked in this `UNICODE_STRING` structure.

The list `EPROCESS` structures is tracked, along with mitigations, the token, and protection level. I'm not currently sure if this is for all processes or just specific ones, as the decompiled output is very mangled. The following process mitigations are checked:

- `AuditDisableDynamicCode`
- `AuditDisallowWin32kSystemCalls`
- `AuditNonSystemFontLoading`
- `AuditFilteredWin32kAPIs`
- `AuditProhibitLowILImageMap`
- `AuditBlockNonMicrosoftBinaries`
- `AuditBlockNonMicrosoftBinariesAllowStore`
- `AuditLoaderIntegrityContinuity`

The following token attributes are checked:

- `Token->IntegrityLevelIndex`
- `Token->UserAndGroups`
- `Token->Lowbox`
- `Token->IntegrityLevel`
- `Token->TokenSource->SourceName`
- `SID->SubAuthorityCount`
- `SID->SubAuthority`

The token of the current `System.exe` process is checked and put into a struct within the script - `L6_2.SystemToken`. The tokens on the system are then enumerated to make sure it's not been stolen.

SgrmLpac.exe

If the assertion engine detects a violation, the SgrmLapc RPC service is used to send an HTTP POST request to `sgrm.microsoft.com/v1.0/Attestation` under the user-agent `SgrmBroker`. A value `atp_onboarded` is included in the requests, if the registry key `SOFTWARE\Microsoft\Windows\CurrentVersion\Sgrm -> Region` is present, this value is sent to the server along with details of the system tampering violation. The region is then used to construct the specific SGRM host, e.g. `europe.sgrm.microsoft.com`, or `unitedkingdom.sgrm.microsoft.com`.

The RPC endpoint can be called by anyone under the IID `a13a9961-953f-4157-8a29-e65e29be510d`, and makes use of the common `Wininet!Http*` family of APIs for the requests. The endpoint has a prototype of (the unknown values don't matter when being called, they can be anything):

```
int s_HttpPost(
    __int64 unk_0,
    PWCHAR pswzServerName,
    PWCHAR *pswcObjectName,
    INTERNET_PORT ipInternetPort,
    PWORD lpHeaders,
    PVOID lpOptional,
    DWORD dwOptionalLength,
    _QWORD *unk_1,
    _DWORD *unk_2
);
```

The RPC call functions end up calling an internal function within the RPC server named appropriately `HttpClient::Post`. James Forshaw's `WindowsRpcClients` here implements this nicely for us.

Conclusion

SGRM is a cool bit of kit, with an interesting architecture. It appears as though the project has gone unmaintaned since 2019, although still running on (just about) all modern devices using Windows Defender. Even though it may be unmaintained, or abandoned, there is still a wealth of potential for checks that could be implemented. The use of Lua allows (or would allow) Microsoft to quickly push new assertions to endpoints, hopefully we can see new updates soon.

Thanks to Alex Ionsecu for his initial research on SGRM, along with helping with queries I had. Along with friends who proof-read this blog (@Myrtus0x00, @herrcore, @smelly__vx, @fwosar, @su_charlie, @hela_luc, @d4n_tweets).

This research was done in personal research time at my own liberty, and does not reflect the opinion of my employer.