wavestone-cdt / EDRSandblast Public

Notifications Fork 278 Star 1.5k

Code Issues 6 Pull requests Actions Projects Security Insights

master Go to file Code

rafaelscheel and themaks Update EDRSandblast_AP... 0710fad · 3 months ago 92 Commits

EDRSandblast	Merge pull request #27 from nuts7/Se...	10 months ago
EDRSandblast_CLI	CLI: bugfix: the output path was too s...	last year
EDRSandblast_LsassDump	visual studio configuration changes	last year
EDRSandblast_StaticLibrary	Update EDRSandblast_API.c - MiniFilter...	3 months ago
Offsets	[new feature] Implements EDR minifilte...	last year
.gitignore	D3FC0N 30 release: Obj callbacks, firew...	2 years ago
EDRSandblast.sln	D3FC0N 30 release: Obj callbacks, firew...	2 years ago
README.md	Completed the README about downlo...	10 months ago

README

EDRSandBlast

`EDRSandBlast` is a tool written in `C` that weaponize a vulnerable signed driver to bypass EDR detections (Notify Routine callbacks, Object Callbacks and `ETW TI` provider) and `LSASS` protections. Multiple userland unhooking techniques are also implemented to evade userland monitoring.

As of release, combination of userland (`--usermode`) and Kernel-land (`--kernelmode`) techniques were used to dump `LSASS` memory under EDR scrutiny, without being blocked nor generating "OS Credential Dumping"-related events in the product (cloud) console. The tests were performed on 3 distinct EDR products and were successful in each case.

Description

EDR bypass through Kernel Notify Routines removal

EDR products use Kernel "Notify Routines" callbacks on Windows to be notified by the kernel of system activity, such as process and thread creation and loading of images (`exe` / `DLL`).

These Kernel callbacks are defined from kernel-land, usually from the driver implementing the callbacks, using a number of documented APIs (`nt!PsSetCreateProcessNotifyRoutine` , `nt!PsSetCreateThreadNotifyRoutine` , etc.). These APIs add driver-supplied callback routines to undocumented arrays of routines in Kernel-space:

- `PspCreateProcessNotifyRoutine` for process creation
- `PspCreateThreadNotifyRoutine` for thread creation
- `PspLoadImageNotifyRoutine` for image loading

About

No description, website, or topics provided.

Readme
Activity
1.5k stars
38 watching
278 forks
Report repository

Releases

No releases published

Packages

No packages published

Contributors 12

Languages

C 89.9% Python 8.4% C++ 1.2% Assembly 0.5%

`EDRSandBlast` enumerates the routines defined in those arrays and remove any callback routine linked to a predefined list of EDR drivers (more than 1000 drivers of security products supported, see the EDR driver detection section. The enumeration and removal are made possible through the exploitation of an arbitrary Kernel memory read / write primitive provided by the exploitation of a vulnerable driver (see Vulnerable drivers section).

The offsets of the aforementioned arrays are recovered using multiple techniques, please refer to Offsets section.

EDR bypass through Object Callbacks removal

EDR (and even EPP) products often register "Object callbacks" through the use of the `nt!ObRegisterCallbacks` kernel API. These callbacks allow the security product to be notified at each handle generation on specific object types (Processes, Threads and Desktops related object callbacks are now supported by Windows). A handle generation may occur on object opening (call to `OpenProcess`, `OpenThread`, etc.) as well as handle duplication (call to `DuplicateHandle`, etc.).

By being notified by the kernel on each of these operations, a security product may analyze the legitimacy of the handle creation (*e.g. an unknown process is trying to open LSASS*), and even block it if a threat is detected.

At each callback registration using `ObRegisterCallbacks`, a new item is added to the `CallbackList` double-linked list present in the `_OBJECT_TYPE` object describing the type of object affected by the callback (either a Process, a Thread or a Desktop). Unfortunately, these items are described by a structure that is not documented nor published in symbol files by Microsoft. However, studying it from various `ntoskrnl.exe` versions seems to indicate that the structure did not change between (at least) Windows 10 builds 10240 and 22000 (from 2015 to 2022).

The mentionned structure, representing an object callback registration, is the following:

```
typedef struct OB_CALLBACK_ENTRY_t {
    LIST_ENTRY CallbackList; // linked element tied to _OBJECT_TYPE.(
    OB_OPERATION Operations; // bitfield : 1 for Creations, 2 for Du
    BOOL Enabled;            // self-explanatory
    OB_CALLBACK* Entry;      // points to the structure in which it
    POBJECT_TYPE ObjectType; // points to the object type affected b
    POB_PRE_OPERATION_CALLBACK PreOperation;      // callback functi
    POB_POST_OPERATION_CALLBACK PostOperation;     // callback funct
    KSPIN_LOCK Lock;         // lock object used for synchronization
} OB_CALLBACK_ENTRY;
```

The `OB_CALLBACK` structure mentionned above is also undocumented, and is defined by the following:

```
typedef struct OB_CALLBACK_t {
    USHORT Version;                             // usually 0x100
    USHORT OperationRegistrationCount;          // number of registere
    PVOID RegistrationContext;                  // arbitrary data pass
    UNICODE_STRING AltitudeString;              // used to determine c
    struct OB_CALLBACK_ENTRY_t EntryItems[1]; // array of OperationR
    WCHAR AltitudeBuffer[1];                    // is AltitudeString.M
} OB_CALLBACK;
```

In order to disable EDR-registered object callbacks, three techniques are implemented in `EDRSandblast`; however only one is enabled for the moment.

Using the `Enabled` field of `OB_CALLBACK_ENTRY`

This is the default technique enabled in `EDRSandblast`. In order to detect and disable EDR-related object callbacks, the `CallbackList` list located in the `_OBJECT_TYPE` objects tied to the *Process* and *Thread* types is browsed. Both `_OBJECT_TYPE`s are pointed by public global symbols in the kernel, `PsProcessType` and `PsThreadType`.

Each item of the list is assumed to fit the `OB_CALLBACK_ENTRY` structure described above (assumption that seems to hold at least in all Windows 10 builds at the time of writing). Functions defined in `PreOperation` and `PostOperation` fields are located to checks if they belong to an EDR driver, and if so, callbacks are simply disabled toggling the `Enabled` flag.

While being a pretty safe technique, it has the inconvenient of relying on an undocumented structure; to reduce the risk of unsafe manipulation of this structure, basic checks are performed to validate that some fields have the expected values :

- `Enabled` is either `TRUE` or `FALSE` (*don't laugh, a* `BOOL` *is an* `int`*, so it could be anything other than* `1` *or* `0`);
- `Operations` is `OB_OPERATION_HANDLE_CREATE`, `OB_OPERATION_HANDLE_DUPLICATE` or both;
- `ObjectType` points on `PsProcessType` or `PsThreadType`.

Unlinking the `CallbackList` of threads and process

Another strategy that do not rely on an undocumented structure (and is thus theoretically more robust against NT kernel changes) is the unlinking of the whole `CallbackList` for both processes and threads. The `_OBJECT_TYPE` object is the following:

```
struct _OBJECT_TYPE {
        LIST_ENTRY TypeList;
        UNICODE_STRING Name;
        [...]
        _OBJECT_TYPE_INITIALIZER TypeInfo;
        [...]
        LIST_ENTRY CallbackList;
}
```

Making the `Flink` and `Blink` pointers of the `CallbackList` `LIST_ENTRY` point to the `LIST_ENTRY` itself effectively make the list empty. Since the `_OBJECT_TYPE` structure is published in the kernel' symbols, the technique does not rely on hardcoded offsets/structures. However, it has some drawbacks.

The first being not able to only disable callbacks from EDR; indeed, the technique affects all object callbacks that could have been registered by "legitimate" software. It should nevertheless be noted that object callbacks are not used by any pre-installed component on Windows 10 (at the time of writing) so disabling them should not affect the machine stability (even more so if the disabling is only temporary).

The second drawback is that process or thread handle operation are really frequent (nearly continuous) in the normal functioning of the OS. As such, if the kernel write primitive used cannot perform a `QWORD` write "atomically", there is a good chance that the `_OBJECT_TYPE.CallbackList.Flink` pointer will be accessed by the kernel in the middle of its overwriting. For instance, the MSI vulnerable driver `RTCore64.sys` can only perform a `DWORD` write at a time, so 2 distinct IOCTLs will be needed to overwrite the pointer, between which the kernel has a high probability of using it (resulting in a crash). On the other hand, the vulnerable DELL driver `DBUtil_2_3.sys` can perform writes of arbitrary sizes in one IOCTL, so using this method with it does not risk causing a crash.

Disabling object callbacks altogether

One last technique we found was to disable entirely the object callbacks support for thread and processes. Inside the `_OBJECT_TYPE` structure corresponding to the process and thread types resides a `TypeInfo` field, following the documented `_OBJECT_TYPE_INITIALIZER` structure. The latter contains a `ObjectTypeFlags` bit field, whose `SupportsObjectCallbacks` flag determines if the described object type (Process, Thread, Desktop, Token, File, etc.) supports object callback registering or not. As previously stated, only Process, Thread and Desktop object types supports these callbacks on a Windows installation at the time of writing.

Since the `SupportsObjectCallbacks` bit is checked by `ObpCreateHandle` or `ObDuplicateObject` before even reading the `CallbackList` (and before executing callbacks, of course), flipping the bit at kernel runtime effectively disable all object callbacks execution.

The main drawback of the method is simply that *KPP* ("*PatchGuard*") monitors the integrity of some (all ?) `_OBJECT_TYPE` structures, and triggers a `0x109 Bug Check` with parameter 4 being equal to `0x8`, meaning an object type structure has been altered.

However, performing the disabling / re-enabling (and "malicious" action in-between) quickly enough should be enough to "race" *PatchGuard* (unless you are unlucky and a periodic check is performed just at the wrong moment).

EDR bypass through minifilters' callbacks unlinking

The Windows Filter Manager system allows an EDR to load a "minifilter" driver and register callbacks in order to be notified of I/O operations, such as file opening, reading, writing, etc.

Here is a quick sum-up of different internal structures used by the filter manager:

- The Filter Manager establishes a "frame" (`_FLTP_FRAME`) as its root structure;
- A "volume" structure (`_FLT_VOLUME`) is instanciated for each "disk" managed by the Filter Manager (can be partitions, shadow copies, or special ones corresponding to named pipes or remote file systems);
- To each registered minifilter driver corresponds a "filter" structure (`_FLT_FILTER`), describing various properties such as its supported operations;
- These minifilters are not all attached to each volume; an "instance" (`_FLT_INSTANCE`) structure is created to mark each of the filter<->volume associations;
- Minifilters register callback functions that are to be executed before and/or after specific operations (file open, write, read, etc.). These callbacks are described in `_CALLBACK_NODE` structures, and can be accessed by different ways:
  - An array of all `_CALLBACK_NODE`s implemented by an instance of a minifilter can be found in the `_FLT_INSTANCE` structure; the array is indexed by the IRP "major function" code, a constant representing the operations handled by the callbacks (`IRP_MJ_CREATE` , `IRP_MJ_READ` , etc.).
  - Also, all `_CALLBACK_NODE`s implemented by instances linked to a specific volume are regrouped in linked lists, stored in the `_FLT_VOLUME.Callbacks.OperationLists` array indexed by IRP major function codes.

These different structures are browsed by `EDRSandblast` to detect filters that are associated with EDR-related drivers, and the callback nodes containing monitoring functions are enumerated. To disable their effect, the nodes are unlinked from their lists, making them temporarily invisible from the filter manager.

This way, during a specified period, the EDR can be completely unaware of any file operations. A basic example would be the creation of an lsass memory dump file on disk, that would not trigger any analysis from the EDR, and thus no detection based on the file itself.

EDR bypass through deactivation of the ETW Microsoft-Windows-Threat-Intelligence provider

The `ETW Microsoft-Windows-Threat-Intelligence` provider logs data about the usages of some Windows API commonly used maliciously. This include the `nt!MiReadWriteVirtualMemory` API, called by `nt!NtReadVirtualMemory` (which is used to dump `LSASS` memory) and monitored by the `nt!EtwTiLogReadWriteVm` function.

EDR products can consume the logs produced by the `ETW TI` provider through services or processes running as, respectively,

`SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT` or `PS_PROTECTED_ANTIMALWARE_LIGHT`, and associated with an `Early Launch Anti Malware (ELAM)` driver.

As published by `slaeryan` in a `CNO Development Labs` blog post, the `ETW TI` provider can be disabled altogether by patching, in kernel memory, its `ProviderEnableInfo` attribute to `0x0`. Refer to the great aforementioned blog post for more information on the technique.

Similarly to the Kernel callbacks removal, the necessary `ntoskrnl.exe` offsets (`nt!EtwThreatIntProvRegHandleOffset`, `_ETW_REG_ENTRY`'s `GuidEntry`, and `_ETW_GUID_ENTRY`'s `ProviderEnableInfo`) are computed in the `NtoskrnlOffsets.csv` file for a number of the Windows Kernel versions.

EDR bypass through userland hooking bypass

How userland hooking works

In order to easily monitor actions that are performed by processes, EDR products often deploy a mechanism called *userland hooking*. First, EDR products register a kernel callback (usually *image loading* or *process creation* callbacks, see above) that allows them to be notified upon each process start.

When a process is loaded by Windows, and before it actually starts, the EDR is able to inject some custom DLL into the process address space, which contains its monitoring logic. While loading, this DLL injects "*hooks*" at the start of every function that is to be monitored by the EDR. At runtime, when the monitored functions are called by the process under surveillance, these hooks redirect the control flow to some supervision code present in the EDR's DLL, which allows it to inspect arguments and return values of these calls.

Most of the time, monitored functions are system calls (such as `NtReadVirtualMemory`, `NtOpenProcess`, etc.), whose implementations reside in `ntdll.dll`. Intercepting calls to `Nt*` functions allows products to be as close as possible to the userland / kernel-land boundary (while remaining in userland), but functions from some higher-level DLLs may also be monitored as well.

Bellow are examples of the same function, before and after beeing hooked by the EDR product:

```
NtProtectVirtualMemory   proc near
        mov r10, rcx
        mov eax, 50h
        test byte ptr ds:7FFE0308h, 1
        jnz short loc_18009D1E5
        syscall
        retn
loc_18009D1E5:
        int 2Eh
        retn
NtProtectVirtualMemory   endp
```

```
NtProtectVirtualMemory proc near
        jmp     sub_7FFC74490298     ; --> "hook", jump to EDR analys
        int 3                        ; overwritten instructions
        int 3                        ; overwritten instructions
        int 3                        ; overwritten instructions
        test byte_7FFE0308, 1        ; <-- execution resumes here af
        jnz short loc_7FFCB44AD1E5
        syscall
        retn
loc_7FFCB44AD1E5:
        int 2Eh
        retn
NtProtectVirtualMemory   endp
```

Hooks detection

Userland hooks have the "weakness" to be located in userland memory, which means they are directly observable and modifiable by the process under scrutiny. To automatically detect hooks in the process address space, the main idea is to compare the differences between the original DLL on disk and the library residing in memory, that has been potentially altered by an EDR. To perform this comparison, the following steps are followed by EDRSandblast:

- The list of all loaded DLLs is enumerated thanks to the `InLoadOrderModuleList` located int the `PEB` (to avoid calling any API that could be monitored and suspicious)
- For each loaded DLL, its content on disk is read and its headers parsed. The corresponding library, residing in memory, is also parsed to identify sections, exports, etc.
- Relocations of the DLL are parsed and applied, by taking the base address of the corresponding loaded library into account. This allows the content of both the in-memory library and DLL originating from disk to have the exact same content (on sections where relocations are applied), and thus making the comparison reliable.
- Exported functions are enumerated and the first bytes of the "in-memory" and "on-disk" versions are compared. Any difference indicates an alteration that has been made after the DLL was loaded, and thus is very probably an EDR hook.

Note: The process can be generalized to find differences anywhere in non-writable sections and not only at the start of exported functions, for example if EDR products start to apply hooks in the middle of function :) Thus not used by the tool, this has been implemented in `findDiffsInNonWritableSections`.

In order to bypass the monitoring performed by these hooks, multiples techniques are possible, and each has benefits and drawbacks.

Hook bypass using ... unhooking

The most intuitive method to bypass the hook-based monitoring is to remove the hooks. Since the hooks are present in memory that is reachable by the process itself, to remove a hook, the process can simply:

- Change the permissions on the page where the hook is located (RX -> RWX or RW)
- Write the original bytes that are known thanks to the on-disk DLL content
- Change back the permissions to RX

This approach is fairly simple, and can be used to remove every detected hook all at once. Performed by an offensive tool at its beginning, this allows the rest of the code to be completely unaware of the hooking mechnanism and perform normally without being monitored.

However, it has two main drawbacks. The EDR is probably monitoring the use of `NtProtectVirtualMemory`, so using it to change the permissions of the page where the hooks have been installed is (at least conceptually) a bad idea. Also, if a thread is executed by the EDR and periodically check the integrity of the hooks, this could also trigger some detection.

For implementation details, check the `unhook()` function's code path when `unhook_method` is `UNHOOK_WITH_NTPROTECTVIRTUALMEMORY`.

Important note: for simplicity, this technique is implemented in EDRSandblast as the base technique used to *showcase* the other bypass techniques; each of them demonstrates how to obtain an unmonitored version of `NtProtectVirtualMemory`, but performs the same operation afterward (unhooking a specific hook).

Hook bypass using a custom trampoline

To bypass a specific hook, it is possible to simply "jump over" and execute the rest of the function as is. First, the original bytes of the monitored function, that have been overwritten by the EDR to install the hook, must be recovered from the DLL file. In our

previous code example, this would be the bytes corresponding to the following instructions:

```
mov r10, rcx
mov eax, 50h
```

Identifying these bytes is a simple task since we are able to perform a clean *diff* of both the memory and disk versions of the library, as previously described. Then, we assemble a jump instruction that is built to redirect the control flow to the code following immediately the hook, at address `NtProtectVirtualMemory + sizeof(overwritten_instructions)`

```
jmp NtProtectVirtualMemory+8
```

Finally, we concatenate these opcodes, store them in (newly) executable memory and keep a pointer to them. This object is called a "*trampoline*" and can then be used as a function pointer, strictly equivalent to the original `NtProtectVirtualMemory` function.

The main benefit of this technique as for every techniques bellow, is that the hook is never erased, so any integrity check performed on the hooks by the EDR should pass. However, it requires to allocate writable then executable memory, which is typical of a shellcode allocation, thus attracting the EDR's scrutiny.

For implementation details, check the `unhook()` function's code path when `unhook_method` is `UNHOOK_WITH_INHOUSE_NTPROTECTVIRTUALMEMORY_TRAMPOLINE`. Please remember the technique is only showcased in our implementation and is, in the end, used to **remove** hooks from memory, as every technique bellow.

Hook bypass using the own EDR's trampoline

The EDR product, in order for its hook to work, must save somewhere in memory the opcodes that it has removed. Worst (*or "better", from the attacker point of view*), to effectively use the original instructions the EDR has probably allocated itself a *trampoline* somewhere to execute the original function after having intercepted the call.

This trampoline can be searched for and used as a replacement for the hooked function, without the need to allocate executable memory, or call any API except `VirtualQuery`, which is most likely not monitored being an innocuous function.

To find the trampoline in memory, we browse the whole address space using `VirtualQuery` looking for commited and executable memory. For each such region of memory, we scan it to look for a jump instruction that targets the address following the overwritten instructions (`NtProtectVirtualMemory+8` in our previous example). The trampoline can then be used to call the hooked function without triggering the hook.

This technique works surprisingly well as it recovers nearly all trampolines on tested EDR. For implementation details, check the `unhook()` function's code path when `unhook_method` is `UNHOOK_WITH_EDR_NTPROTECTVIRTUALMEMORY_TRAMPOLINE`.

Hook bypass using duplicate DLL

Another simple method to get access to an unmonitored version of `NtProtectVirtualMemory` function is to load a duplicate version of the `ntdll.dll` library into the process address space. Since two identical DLLs can be loaded in the same process, provided they have different names, we can simply copy the legitimate `ntdll.dll` file into another location, load it using `LoadLibrary` (or reimplement the loading process), and access the function using `GetProcAddress` for example.

This technique is very simple to understand and implement, and have a decent chance of success, since most of EDR products does not re-install hooks on newly loaded DLLs once the process is running. However, the major drawback is that copying Microsoft signed binaries under a different name is often considered as suspicious by EDR products as itself.

This technique is nevertheless implemented in `EDRSandblast`. For implementation details, check the `unhook()` function's code path when `unhook_method` is `UNHOOK_WITH_DUPLICATE_NTPROTECTVIRTUALMEMORY`.

Hook bypass using direct syscalls

In order to use system calls related functions, one program can reimplement syscalls (in assembly) in order to call the corresponding OS features without actually touching the code in `ntdll.dll`, which might be monitored by the EDR. This completely bypasses any userland hooking done on syscall functions in `ntdll.dll`.

This nevertheless has some drawbacks. First, this implies being able to know the list of syscall numbers of functions the program needs, which changes for each version of Windows. This is nevertheless mitigated by implementing multiple heuristics that are known to work in all the past versions of Windows NT (sorting `ntdll`'s' `Zw*` exports, searching for `mov rax, #syscall_number` instruction in the associated `ntdll` function, etc.), and checking they all return the same result (see `Syscalls.c` for more details).

Also, functions that are not technically syscalls (e.g. `LoadLibraryX` / `LdrLoadDLL`) could be monitored as well, and cannot simply be reimplemented using a syscall.

The direct syscalls technique is implemented in EDRSandblast. As previously stated, it is only used to execute `NtProtectVirtualMemory` safely, and remove all detected hooks.

For implementation details, check the `unhook()` function's code path when `unhook_method` is `UNHOOK_WITH_DIRECT_SYSCALL`.

Vulnerable drivers exploitation

As previously stated, every action that needs a kernel memory read or write relies on a vulnerable driver to give this primitive. In EDRSanblast, adding the support for a new driver providing the read/write primitive can be "easily" done, only three functions need to be implemented:

- A `ReadMemoryPrimitive_DRIVERNAME(SIZE_T Size, DWORD64 Address, PVOID Buffer)` function, that copies `Size` bytes from kernel address `Address` to userland buffer `Buffer`;
- A `WriteMemoryPrimitive_DRIVERNAME(SIZE_T Size, DWORD64 Address, PVOID Buffer)` function, that copies `Size` bytes from userland buffer `Buffer` to kernel address `Address`;
- A `CloseDriverHandle_DRIVERNAME()` that ensures all handles to the driver are closed (needed before uninstall operation which is driver-agnostic, for the moment).

As an example, two drivers are currently supported by EDRSandblast, `RTCore64.sys` (SHA256: `01AA278B07B58DC46C84BD0B1B5C8E9EE4E62EA0BF7A695862444AF32E87F1FD`) and `DBUtils_2_3.sys` (SHA256: `0296e2ce999e67c76352613a718e11516fe1b0efc3ffdb8918fc999dd76a73a5`). The following code in `KernelMemoryPrimitives.h` is to be updated if the used vulnerable driver needs to be changed, or if a new one implemented.

```
#define RTCore 0
#define DBUtil 1
// Select the driver to use with the following #define
#define VULN_DRIVER RTCore

#if VULN_DRIVER == RTCore
#define DEFAULT_DRIVER_FILE TEXT("RTCore64.sys")
#define CloseDriverHandle CloseDriverHandle_RTCore
#define ReadMemoryPrimitive ReadMemoryPrimitive_RTCore
#define WriteMemoryPrimitive WriteMemoryPrimitive_RTCore
#elif VULN_DRIVER == DBUtil
#define DEFAULT_DRIVER_FILE TEXT("DBUtil_2_3.sys")
#define CloseDriverHandle CloseDriverHandle_DBUtil
#define ReadMemoryPrimitive ReadMemoryPrimitive_DBUtil
```

```
#define WriteMemoryPrimitive WriteMemoryPrimitive_DBUtil
#endif
```

EDR drivers and processes detection

Multiple techniques are currently used to determine if a specific driver or process belongs to an EDR product or not.

First, the name of the driver can simply be used for that purpose. Indeed, Microsoft allocates specific numbers called "Altitudes" for all drivers that need to insert callbacks in the kernel. This allow a deterministic order in callbacks execution, independent from the registering order, but only based on the driver usage. A list of (vendors of) drivers that have reserved specific *altitude* can be found on MSDN. As a consequence, a nearly comprehensive list of security driver names tied to security products is offered by Microsoft, mainly in the "FSFilter Anti-Virus" and "FSFilter Activity Monitor" lists. These lists of driver names are embedded in EDRSandblast, as well as additional contributions.

Moreover, EDR executables and DLL are more than often digitally signed using the vendors signing certificate. Thus, checking the signer of an executable or DLL associated to a process may allow to quickly identify EDR products.

Also, drivers need to be directly signed by Microsoft to be allowed to be loaded in kernel space. While the driver's vendor is not directly the signer of the driver itself, it would seam that the vendor's name is still included inside an attribute of the signature; this detection technique is nevertheless yet to be investigated and implemented.

Finally, when facing an EDR unknown to EDRSandblast, the best approach is to run the tool in "audit" mode, and check the list of drivers having registered kernel callbacks; then the driver's name can be added to the list, the tool recompiled and re-run.

RunAsPPL bypass

The `Local Security Authority (LSA) Protection` mechanism, first introduced in Windows 8.1 and Windows Server 2012 R2, leverage the `Protected Process Light (PPL)` technology to restrict access to the `LSASS` process. The `PPL` protection regulates and restricts operations, such as memory injection or memory dumping of protected processes, even from a process holding the `SeDebugPrivilege` privilege. Under the process protection model, only processes running with higher protection levels can perform operations on protected processes.

The `_EPROCESS` structure, used by the Windows kernel to represent a process in kernel memory, includes a `_PS_PROTECTION` field defining the protection level of a process through its `Type` (`_PS_PROTECTED_TYPE`) and `Signer` (`_PS_PROTECTED_SIGNER`) attributes.

By writing in kernel memory, the EDRSandblast process is able to upgrade its own protection level to `PsProtectedSignerWinTcb-Light`. This level is sufficient to dump the `LSASS` process memory, since it "dominates" to `PsProtectedSignerLsa-Light`, the protection level of the `LSASS` process running with the `RunAsPPL` mechanism.

`EDRSandBlast` implements the self protection as follow:

- open a handle to the current process
- leak all system handles using `NtQuerySystemInformation` to find the opened handle on the current process, and the address of the current process' `EPROCESS` structure in kernel memory.
- use the arbitrary read / write vulnerability of the vulnerable driver to overwrite the `_PS_PROTECTION` field of the current process in kernel memory. The offsets of the `_PS_PROTECTION` field relative to the `EPROCESS` structure (defined by the `ntoskrnl` version in use) are computed in the `NtoskrnlOffsets.csv` file.

Credential Guard bypass

Microsoft `Credential Guard` is a virtualization-based isolation technology, introduced in Microsoft's `Windows 10 (Enterprise edition)` which prevents direct access to the credentials stored in the `LSASS` process.

When `Credentials Guard` is activated, an `LSAIso` (*LSA Isolated*) process is created in `Virtual Secure Mode`, a feature that leverages the virtualization extensions of the CPU to provide added security of data in memory. Access to the `LSAIso` process are restricted even for an access with the `NT AUTHORITY\SYSTEM` security context. When processing a hash, the `LSA` process perform a `RPC` call to the `LSAIso` process, and waits for the `LSAIso` result to continue. Thus, the `LSASS` process won't contain any secrets and in place will store `LSA Isolated Data`.

As stated in original research conducted by `N4kedTurtle`: "`Wdigest` can be enabled on a system with Credential Guard by patching the values of `g_fParameter_useLogonCredential` and `g_IsCredGuardEnabled` in memory". The activation of `Wdigest` will result in cleartext credentials being stored in `LSASS` memory for any new interactive logons (without requiring a reboot of the system). Refer to the original research blog post for more details on this technique.

`EDRSandBlast` simply make the original PoC a little more opsec friendly and provide support for a number of `wdigest.dll` versions (through computed offsets for `g_fParameter_useLogonCredential` and `g_IsCredGuardEnabled`).

Offsets retrieval

In order to reliably perform kernel monitoring bypass operations, EDRSandblast needs to know exactly where to read and write kernel memory. This is done using offsets of global variables inside the targeted image (ntoskrnl.exe, wdigest.dll), as well as offset of specific fields in structures whose definitions are published by Microsoft in symbol files. These offsets are specific to each build of the targeted images, and must be gathered at least once for a specific platform version.

The choice of using "hardcoded" offsets instead of pattern searches to locate the structures and variables used by EDRSandblast is justified by the fact that the undocumented APIs responsible for Kernel callbacks addition / removal are subject to change and that any attempt to read or write Kernel memory at the wrong address may (and often will) result in a `Bug Check` (`Blue Screen of Death`). A machine crash is not acceptable in both red-teaming and normal penetration testing scenarios, since a machine that crashes is highly visible by defenders, and will lose any credentials that was still in memory at the moment of the attack.

To retrieve offsets for each specific version of Windows, two approaches are implemented.

Manual offset retrieval

The required `ntoskrnl.exe` and `wdigest.dll` offsets can be extracted using the provided `ExtractOffsets.py` Python script, that relies on `radare2` and `r2pipe` to download and parse symbols from PDB files, and extracted the needed offsets from them. Offsets are then stored in CSV files for later use by EDRSandblast.

In order to support out-of-the-box a wide range of Windows builds, many versions of the `ntoskrnl.exe` and `wdigest.dll` binaries are referenced by Winbindex, and can be automatically downloaded (and their offsets extracted) by the `ExtractOffsets.py`. This allows to extract offsets from nearly all files that were ever published in Windows update packages (to date 450+ `ntoskrnl.exe` and 30+ `wdigest.dll` versions are available and pre-computed).

Automatic offsets retrieval and update

An additionnal option has been implemented in `EDRSandBlast` to allow the program to download the needed `.pdb` files itself from Microsoft Symbol Server, extract the required offsets, and even update the corresponding `.csv` files if present.

Using the `--internet` option make the tool execution much simpler, while introducing an additionnal OpSec risk, since a `.pdb` file is downloaded and dropped on disk during the process. This is required by the `dbghelp.dll` functions used to parse the symbols database ; however, full in-memory PDB parsing might be implemented in the future to lift this requirement and reduce the tool's footprint.

Usage

Vulnerable drivers

EDRSandblast publicly implements the support of at least 3 vulnerable driver, `gdrv.sys` (default), `RTCore64.sys` and `DBUtil_2_3.sys`. The driver actually used is decided before compilation of the tool (see `#define VULN_DRIVER <driver name>` in `includes/KernelMemoryPrimitive.h`). A copy of the vulnerable driver should be downloaded and provided to EDRSandblast for its kernel operation to work.

Tested drivers' hashs are mentionned at the start of each `Driver<name>.c` file that implements the kernel memory read and write primitives used by EDRSanblast. Using these hashs, drivers samples can be easy found on the Internet, especially on `https://www.loldrivers.io`.

Here is the list of the supported vulnerable drivers along with download links:

Supported driver	Download link	SHA256
`GDRV.sys`	LOLDrivers link	31f4cfb4c71da44120752721103a16512444c13
`RTCore64.sys`	LOLDrivers link	01aa278b07b58dc46c84bd0b1b5c8e9ee4e62ea
`DBUtil_2_3.sys`	LOLDrivers link	0296e2ce999e67c76352613a718e11516fe1b0e

Quick usage

```
Usage: EDRSandblast.exe [-h | --help] [-v | --verbose] <audit | dump
[--usermode] [--unhook-method <N>] [--direct-syscalls] [--add-dll <d
[--kernelmode] [--dont-unload-driver] [--no-restore]
    [--nt-offsets <NtoskrnlOffsets.csv>] [--fltmgr-offsets <FltmgrOf
    [--vuln-driver <RTCore64.sys>] [--vuln-service <SERVICE_NAME>]
    [--unsigned-driver <evil.sys>] [--unsigned-service <SERVICE_NAME
    [--no-kdp]
[-o | --dump-output <DUMP_FILE>]
```

Options

```
-h | --help             Show this help message and exit.
-v | --verbose          Enable a more verbose output.

Actions mode:

        audit                   Display the user-land hooks and / o
        dump                    Dump the process specified by --pro
                                specified file using -o | --output
        cmd                     Open a cmd.exe prompt.
        credguard               Patch the LSASS process' memory to
                                Credential Guard is enabled on the
        firewall                Add Windows firewall rules to block
        load_unsigned_driver    Load the specified unsigned driver
                                WARNING: currently an experimental

--usermode              Perform user-land operations (DLL unhooking)
--kernelmode            Perform kernel-land operations (Kernel callba
```

```
Hooking-related options:

--add-dll <dll name or path>            Loads arbitrary libraries in
                                        anything.This can be useful
                                        loaded by default by this pr
                                        multiple DLLs all at once.
                                        Example of interesting DLLs
                                        samcli.dll, winhttp.dll, urlm

--unhook-method <N>                     Choose the userland un-hooki

        0                               Do not perform any unhooking
        1 (Default)                     Uses the (probably monitored
                                        present userland hooks.
        2                               Constructs a 'unhooked' (i.e
                                        userland hooks.
        3                               Searches for an existing tra
                                        (i.e. unmonitored) version o
                                        hooks.
        4                               Loads an additional version
                                        present userland hooks.
        5                               Allocates a shellcode that u

--direct-syscalls       Use direct syscalls to dump the selected pro


BYOVD options:

--dont-unload-driver                    Keep the vulnerable driver i
                                        Default to automatically uns
--no-restore                            Do not restore the EDR drive
                                        Default to restore the callb
--vuln-driver <gdrv.sys>                Path to the vulnerable drive
                                        Default to 'gdrv.sys' in the
--vuln-service <SERVICE_NAME>           Name of the vulnerable servi


Driver sideloading options:

--unsigned-driver <evil.sys>            Path to the unsigned driver
                                        Default to 'evil.sys' in the
--unsigned-service <SERVICE_NAME>       Name of the unsigned driver'
--no-kdp                                Switch to g_CiOptions patchi


Offset-related options:

--nt-offsets <NtoskrnlOffsets.csv>      Path to the CSV file contain
                                        Default to 'NtoskrnlOffsets.
--fltmgr-offsets <FltmgrOffsets.csv>    Path to the CSV file contain
                                        Default to 'FltmgrOffsets.cs
--wdigest-offsets <WdigestOffsets.csv>  Path to the CSV file contain
                                        (only for the 'credguard' mo
                                        Default to 'WdigestOffsets.c
--ci-offsets <CiOffsets.csv>            Path to the CSV file contain
                                        (only for the 'load_unsigned
                                        Default to 'WdigestOffsets.c
-i | --internet                         Enables automatic symbols do
                                        If a corresponding *Offsets.
                                        OpSec warning: downloads and


Dump options:

-o | --dump-output <DUMP_FILE>          Output path to the dump file
                                        Default to 'process_name' in
--process-name <NAME>                   File name of the process to
```

Build

`EDRSandBlast` (x64 only) was built on Visual Studio 2019 (Windows SDK Version: `10.0.19041.0` and Plateform Toolset: `Visual Studio 2019 (v142)`).

ExtractOffsets.py usage

Note that `ExtractOffsets.py` has only be tested on Windows.

```
# Installation of Python dependencies
pip.exe install -m .\requirements.txt

# Script usage
ExtractOffsets.py [-h] -i INPUT [-o OUTPUT] [-d] mode

positional arguments:
  mode                  ntoskrnl or wdigest. Mode to download and ex

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Single file or directory containing ntoskrnl
                        If in download mode, the PE downloaded from
  -o OUTPUT, --output OUTPUT
                        CSV file to write offsets to. If the specifi
                        downloaded / analyzed.
                        Defaults to NtoskrnlOffsets.csv / WdigestOff
  -d, --download         Flag to download the PE from Microsoft serv
```

Detection

From the defender (EDR vendor, Microsoft, SOC analysts looking at EDR's telemetry, ...) point of view, multiple indicators can be used to detect or prevent this kind of techniques.

Driver whitelisting

Since every action performed by the tool in kernel-mode memory relies on a vulnerable driver to read/write arbitrary content, driver loading events should be heaviliy scrutinized by EDR product (or SOC analysts), and raise an alert at any uncommon driver loading, or even block known vulnerable drivers. This latter approach is even recommended by Microsoft themselves: any HVCI (*Hypervisor-protected code integrity*) enabled Windows device embeds a drivers blocklist, and this will be progressively become a default behaviour on Windows (it already is on Windows 11).

Kernel-memory integrity checks

Since an attacker could still use an unknown vulnerable driver to perform the same actions in memory, the EDR driver could periodically check that its kernel callbacks are still registered, directly by inspecting kernel memory (like this tool does), or simply by triggering events (process creation, thread creation, image loading, etc.) and checking the callback functions are indeed called by the executive kernel.

As a side note, this type of data structure could be protected via the recent Kernel Data Protection (KDP) mechanism, which relies on Virtual Based Security, in order to make the kernel callbacks array non-writable without calling the right APIs.

The same logic could apply to sensitive ETW variables such as the `ProviderEnableInfo`, abused by this tool to disable the ETW Threat Intelligence events generation.

User-mode detection

The first indicator that a process is actively trying to evade user-land hooking is the file accesses to each DLL corresponding to loaded modules; in a normal execution, a userland process rarely needs to read DLL files outside of a `LoadLibrary` call, especially `ntdll.dll`.

In order to protect API hooking from being bypassed, EDR products could periodically check that hooks are not altered in memory, inside each monitored process.

Finally, to detect hooking bypass (abusing a trampoline, using direct syscalls, etc.) that does not imply the hooks removal, EDR products could potentially rely on kernel

callbacks associated to the abused syscalls (ex. `PsCreateProcessNotifyRoutine` for `NtCreateProcess` syscall, `ObRegisterCallbacks` for `NtOpenProcess` syscall, etc.), and perform user-mode call-stack analysis in order to determine if the syscall was triggered from a normal path (`kernel32.dll` -> `ntdll.dll` -> syscall) or an abnormal one (ex. `program.exe` -> direct syscall).

Acknowledgements

- Kernel callbacks enumeration and removal: https://github.com/br-sn/CheekyBlinder
- Kernel memory Read / Write primitives through the vulnerable `Micro-Star MSI Afterburner` driver: https://github.com/Barakat/CVE-2019-16098/
- Disabling of the ETW Threat Intelligence provider: https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/exploits/data-only-attack-neutralizing-etwti-provider
- Driver install / uninstall: https://github.com/gentilkiwi/mimikatz
- Initial list of EDR drivers names: https://github.com/SadProcessor/SomeStuff/blob/master/Invoke-EDRCheck.ps1
- Credential Guard bypass by re-enabling `Wdigest` through `LSASS` memory patching: https://teamhydra.blog/2020/08/25/bypassing-credential-guard/

Authors

Thomas DIOT (Qazeer) Maxime MEIGNAN (themaks)

Thanks to contributors

- v1k1ngfr: for Driver Signature Enforcement bypass (via `g_CiOptions` patching) and GDRV.sys driver support