An End to KASLR Bypasses?

windows-internals.com/an-end-to-kaslr-bypasses

By Yarden Shafir

Edit: this post initially discussed the new changes only in the context of KASLR bypasses. In reality this new event covers other suspicious behaviors as well and the post was edited to reflect that. The title is left as it was for convenience.

In recent years, in addition to mitigating and patching specific malware or exploits, Microsoft is targeting bug classes. With a wide range of mitigations, such as zero-initialized pool allocations, CET, XFG and the most recent CastGuard, exploiting bugs is becoming more and more challenging. On top of that, there is improved visibility into malware and exploit techniques through ETW and specifically the Threat Intelligence ETW channel, available to EDRs.

In `23H2` preview builds, Microsoft is introducing a new ETW event, this time aimed at NT APIs that could point at various suspicious behaviors.

Syscall Usage Visibility

With this new change, Microsoft is focusing on several system calls that normally shouldn't be used by many applications but might be used by exploits either in their pre- or post-exploitation stage for various purposes, such as KASLR bypasses, VM detection or physical memory access. Many of the cases covered by this new event are already restricted to privileged processes — some require privileges reserved to admin or system processes, others restricted to low IL or untrusted callers. But an attempt to call any of those system calls could indicate suspicious activity, so it could be interesting regardless.

Until now, the only way EDRs could detect this type of activity was to place user-mode hooks on all the different `NtQuery` functions that leak kernel pointers. For many reasons, this is not ideal. Microsoft has been trying to keep EDRs away from user-mode hooks for a while, mostly by adding ETW events that allow EDRs to consume the same information through non-invasive means (though asynchronously and with no blocking capabilities).

Keeping up with this trend, Windows 11 `23H2` adds a new ETW event to the Threat Intelligence channel – `THREATINT_PROCESS_SYSCALL_USAGE`. This ETW event is generated to indicate that a non-admin process has made an API call to an API + information class that could indicate some unusual (and potentially malicious) activity. This event will be generated for information classes in two APIs:

- `NtQuerySystemInformation`

- `NtSystemDebugControl`

These APIs have many information classes and many of them are “innocent” and commonly used by many applications. To avoid spamming information that isn’t interesting or useful, the following information classes will generate an ETW event:

- `SystemModuleInformation`
- `SystemModuleInformationEx`
- `SystemLocksInformation`
- `SystemStackTraceInformation`
- `SystemHandleInformation`
- `SystemExtendedHandleInformation`
- `SystemObjectInformation`
- `SystemBigPoolInformation`
- `SystemExtendedProcessInformation`
- `SystemSessionProcessInformation`
- `SystemMemoryTopologyInformation`
- `SystemMemoryChannelInformation`
- `SystemCoverageInformation`
- `SystemPlatformBinaryInformation`
- `SystemFirmwareTableInformation`
- `SystemBootMetadataInformation`
- `SystemWheaIpmiHardwareInformation`
- `SystemSuperfetchInformation + SuperfetchPrefetch`
- `SystemSuperfetchInformation + SuperfetchPfnQuery`
- `SystemSuperfetchInformation + SuperfetchPrivSourceQuery`
- `SystemSuperfetchInformation + SuperfetchMemoryListQuery`
- `SystemSuperfetchInformation + SuperfetchMemoryRangesQuery`
- `SystemSuperfetchInformation + SuperfetchPfnSetPriority`
- `SystemSuperfetchInformation + SuperfetchMovePages`
- `SystemSuperfetchInformation + SuperfetchPfnSetPageHeat`
- `SysDbgGetTriageDump`
- `SysDbgGetLiveKernelDump`

These information classes are included for different reasons – some are known to leak kernel addresses, some can be used for VM detection, another used in hardware persistence, and some indicate previous knowledge of physical memory that most applications should not have. Overall, this new event covers various indicators that an application isn’t behaving as it should.

Every mitigation must also take into consideration the potential performance impact, and ETW event generation can slow down the system when done in a code path that is called frequently. So, a few restrictions apply to this:

1. The events will only be generated for user-mode non-admin callers. Since Admin->Kernel is not considered a boundary on Windows, many mitigations don't apply to admin processes to lower the performance impact on the system.
2. An event will only be generated once per information class for each process. This means if `NtQuerySystemInformation` is called 10 times by a single process, all with the same information class, only one ETW event will be sent.
3. The event will only be sent if the call succeeded. Failed calls will be ignored and will not generate any events.

To support requirement `2` and keep track of which information class were involved by a process, a new field was added to the `EPROCESS` structure:

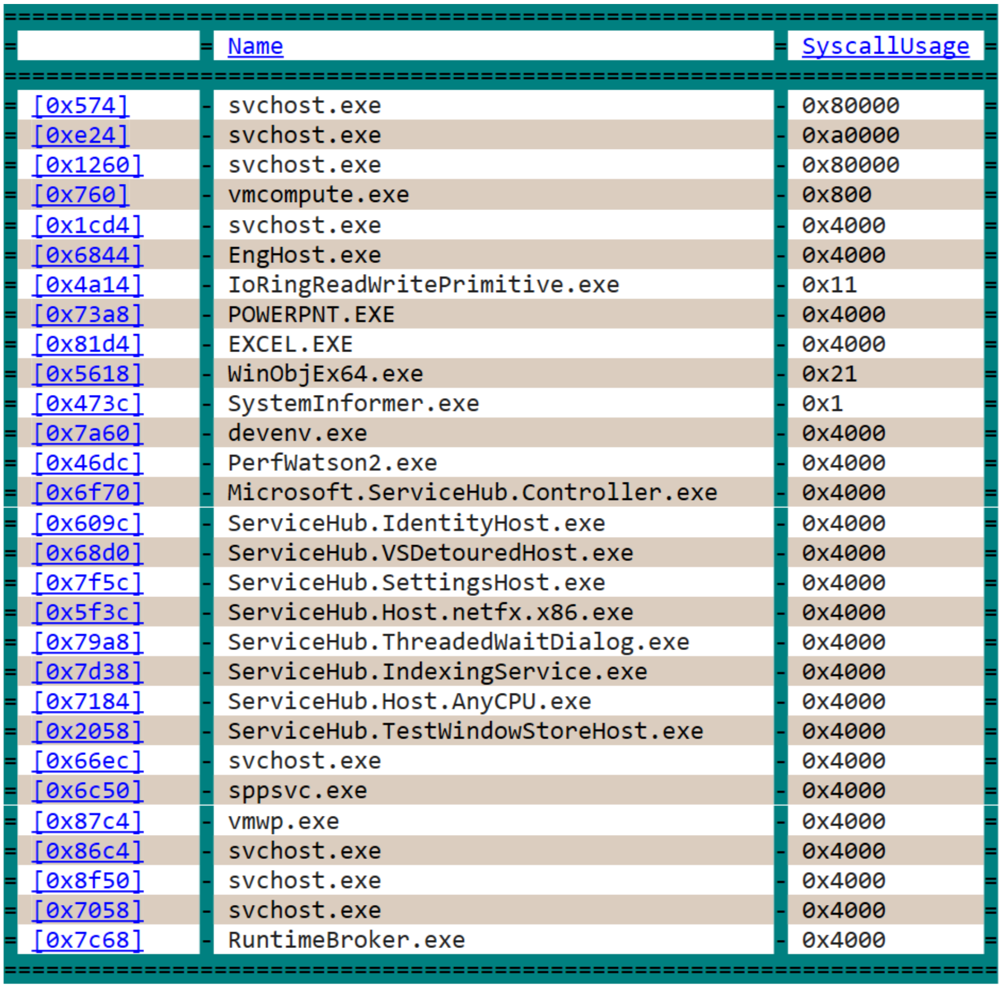
```
union
{
    unsigned long SyscallUsage;
    struct
    {
        struct /* bitfield */
        {
            unsigned long SystemModuleInformation : 1; /* bit position: 0 */
            unsigned long SystemModuleInformationEx : 1; /* bit position: 1 */
            unsigned long SystemLocksInformation : 1; /* bit position: 2 */
            unsigned long SystemStackTraceInformation : 1; /* bit position: 3 */
            unsigned long SystemHandleInformation : 1; /* bit position: 4 */
            unsigned long SystemExtendedHandleInformation : 1; /* bit position: 5 */
            unsigned long SystemObjectInformation : 1; /* bit position: 6 */
            unsigned long SystemBigPoolInformation : 1; /* bit position: 7 */
            unsigned long SystemExtendedProcessInformation : 1; /* bit position: 8 */
            unsigned long SystemSessionProcessInformation : 1; /* bit position: 9 */
            unsigned long SystemMemoryTopologyInformation : 1; /* bit position: 10 */
            unsigned long SystemMemoryChannelInformation : 1; /* bit position: 11 */
            unsigned long SystemCoverageInformation : 1; /* bit position: 12 */
            unsigned long SystemPlatformBinaryInformation : 1; /* bit
position: 13 */
             unsigned long SystemFirmwareTableInformation : 1; /* bit
position: 14 */
             unsigned long SystemBootMetadataInformation : 1; /* bit
position: 15 */
             unsigned long SystemWheaIpmiHardwareInformation : 1; /* bit
position: 16 */
             unsigned long SystemSuperfetchPrefetch : 1; /* bit position: 17
*/
             unsigned long SystemSuperfetchPfnQuery : 1; /* bit position: 18
*/
             unsigned long SystemSuperfetchPrivSourceQuery : 1; /* bit
position: 19 */
             unsigned long SystemSuperfetchMemoryListQuery : 1; /* bit
position: 20 */
             unsigned long SystemSuperfetchMemoryRangesQuery : 1; /* bit
position: 21 */
             unsigned long SystemSuperfetchPfnSetPriority : 1; /* bit
position: 22 */
             unsigned long SystemSuperfetchMovePages : 1; /* bit position:
23 */
             unsigned long SystemSuperfetchPfnSetPageHeat : 1; /* bit
position: 24 */
             unsigned long SysDbgGetTriageDump : 1; /* bit position: 25 */
             unsigned long SysDbgGetLiveKernelDump : 1; /* bit position: 26
*/
             unsigned long SyscallUsageValuesSpare : 5; /* bit position: 27
*/
         }; /* bitfield */
     }  SyscallUsageValues;
 };
```

The first time a process successfully invokes one of the monitored information classes, the bit corresponding to that information class is set – this happens for admin processes, even if the ETW event isn't sent for those processes. An ETW event is only sent if the bit is not set, guaranteeing that an event is only sent once for every class. And while there is no API to query this `EPROCESS` field, it does have the nice side effect of leaving a record of which information classes are used by each process – something to look at if you analyze a system! (But only if the Syscall Usage event is enabled in the system, otherwise the bits don't get set).

Examining the Data

Currently nothing is enabling this event, and no one consumes it, but I expect to see Windows Defender start using it soon, and hopefully other EDRs as well. I went and enabled this event manually to see whether those "suspicious" APIs get used on a regular machine, using my I/O ring exploit as a sanity test (since I know it uses `NtQuerySystemInformation` to leak kernel pointers). Here are some of the results from a few minutes of normal execution:

```
dx -g @$cursession.Processes.Where(p =>
p.KernelObject.SyscallUsage).Select(p => new {Name = p.Name, SyscallUsage =
p.KernelObject.SyscallUsage})
```



	Name	SyscallUsage
[0x574]	svchost.exe	0x80000
[0xe24]	svchost.exe	0xa0000
[0x1260]	svchost.exe	0x80000
[0x760]	vmcompute.exe	0x800
[0x1cd4]	svchost.exe	0x4000
[0x6844]	EngHost.exe	0x4000
[0x4a14]	IoRingReadWritePrimitive.exe	0x11
[0x73a8]	POWERPNT.EXE	0x4000
[0x81d4]	EXCEL.EXE	0x4000
[0x5618]	WinObjEx64.exe	0x21
[0x473c]	SystemInformer.exe	0x1
[0x7a60]	devenv.exe	0x4000
[0x46dc]	PerfWatson2.exe	0x4000
[0x6f70]	Microsoft.ServiceHub.Controller.exe	0x4000
[0x609c]	ServiceHub.IdentityHost.exe	0x4000
[0x68d0]	ServiceHub.VSDetouredHost.exe	0x4000
[0x7f5c]	ServiceHub.SettingsHost.exe	0x4000
[0x5f3c]	ServiceHub.Host.netfx.x86.exe	0x4000
[0x79a8]	ServiceHub.ThreadedWaitDialog.exe	0x4000
[0x7d38]	ServiceHub.IndexingService.exe	0x4000
[0x7184]	ServiceHub.Host.AnyCPU.exe	0x4000
[0x2058]	ServiceHub.TestWindowStoreHost.exe	0x4000
[0x66ec]	svchost.exe	0x4000
[0x6c50]	sppsvc.exe	0x4000
[0x87c4]	vmwp.exe	0x4000
[0x86c4]	svchost.exe	0x4000
[0x8f50]	svchost.exe	0x4000
[0x7058]	svchost.exe	0x4000
[0x7c68]	RuntimeBroker.exe	0x4000

Obviously, there are a few information classes that are used pretty frequently on the machine, with the main one (so far) being `SystemFirmwareTableInformation`. Those common classes might get ignored by EDRs early on, and therefore become more popular with exploits that will be able to abuse them. Other classes are not as common and are more unique to exploits, though valid software may use it as well, resulting in false detections.

Conclusion

Does this mean there are no more API-based KASLR bypasses? Or that all existing exploits will immediately get detected? Probably not. EDRs will take a while to start registering for these events and using them, especially since `23H2` will only be officially released some time next fall and it'll probably be another year or two until most security products realize this event exists. And since this event is sent to the Threat Intelligence channel, which only PPLs can register for, many products can't access this or other exploit-related events at all. Besides, even for the security products that will register for this event, this isn't a world-changing addition. This ETW event simply replaces a few user-mode hooks that some EDRs were already using, without supplying entirely new capabilities. This event will enable EDRs to get information for some additional calls done by malicious processes, but that is only a single step in an exploit and will undoubtedly lead to many false positive if security products rely on it too heavily. And anyway, this event only covers some known indicators, leaving many others as potential bypasses

To summarize, this is a cool addition that I hope security products will use to add another layer of visibility into potential exploits. While it's not a game changer just yet, it's definitely something for both EDRs and exploit developers to consider in the near future.