Windows ARM64 Internals: Deconstructing Pointer Authentication

originhq.com/research/windows-arm64-internals-deconstructing-pointer-authentication

Connor McGarr October 13, 2025

← Back to Research

Pointer Authentication Code, or PAC, is an anti-exploit/memory-corruption feature that signs pointers so their use (as code or data) can be validated at runtime. PAC is available on Armv8.3-A and Armv9.0-A (and later) ARM architectures and leverages virtual addressing in order to store a small cryptographic signature alongside the pointer value.

On a typical 64-bit processor a pointer is considered a "user-mode" pointer if bit 47 of a 64-bit address is set to 0 (meaning, then, bits 48-63 are also 0). This is known as a *canonical* user-mode address. If bit 47 is set to 1, bits 48-63 are also set to 1, with this being considered a canonical *kernel-mode* address. Additionally, LA57, ARM 52 or 56 bit, or similar processors extend the most significant bit out even further (and PAC can also be enabled in the ARM-specific scenarios). For our purposes, however, we will be looking at a typical 64-bit processor with the most significant bit being bit 47.

It has always been an "accepted" standard that the setting of the most significant bit denotes a user-mode or kernel-mode address – with even some hardware vendors, like Intel, formalizing this architecture in actual hardware with CPU features like Linear Address Space Separation (LASS). This means that bits 48-63 are unused on a current, standard 64-bit processor, as the OS typically ignores them. Because they are unused, this allows PAC to store the aforementioned signature in these unused bits alongside the pointer itself.

As mentioned, these “unused” bits are now used to store signing information about a particular pointer in order to validate and verify execution and/or data access to the target memory address. Special CPU instructions are used to both generate and validate cryptographic signatures associated with a particular pointer value. This blog post will examine the Windows implementation of PAC on ARM64 installations of Windows, which, as we will see, supports a very specific implementation of PAC in both user-mode and kernel-mode.

PAC Enablement on Windows

PAC enablement on Windows begins at the entry point of ntoskrnl.exe, KiSystemStartup. KiSystemStartup is responsible for determining if PAC is supported on Windows and also for initializing basic PAC support. KiSystemStartup receives the *loader parameter block* (LOADER_PARAMETER_BLOCK) from winload.efi, the Windows boot loader. The loader block denotes if PAC is supported. Specifically, the *loader parameter block extension* (LOADER_PARAMETER_EXTENSION) portion of the loader block defines a bitmask of various features which are present/supported, so say the boot loader. The PointerAuthKernelIpEnabled bit of this bitmask denotes if PAC is supported. If PAC is supported, the loader parameter block extension is also responsible for providing the initial PAC signing key (PointerAuthKernelIpKey) used to sign and authenticate all kernel-mode pointers (we will see later that the "current" signing key is updated many times). When execution is occurring in kernel-mode, this is the key used to sign kernel-mode pointers. The bootloader generates the key in OslPrepareTarget by calling the function SymCryptRngAesGenerate to generate the initial kernel pointer signing key passed via the loader parameter block.

```
if ( pointerAuthSupported )
{
  isQarmaEnabled = _ReadStatusReg(ID_AA64ISAR1_EL1) >> 4;
  if ( ((_ReadStatusReg(ID_AA64ISAR1_EL1) >> 8) & 0xF) != 0 || isQarmaEnabled != 0 )
  {
    loaderParameterBlock->Extension->LoaderParameterFlags |= 0x10000u;
    SymCryptRngAesGenerate(&SymCryptRng, &loaderParameterBlock->Extension->PointerAuthKernelIpKey, 0x10u);
  }
}
```

The ARM architecture supports having *multiple* signing keys for different scenarios, like signing instruction pointers or data pointers with *different* keys. Typically, "key A" and "key B" (as they are referred to), which are stored in specific system registers, are used for signing pointers used in instruction executions (like return addresses). Windows currently only uses PAC for "instruction pointers" (more on this later) and it also it only uses "key B" for cryptographic signatures and, therefore, loads the target pointer signing value into the APIBKeyLo_EL1 and APIBKeyHi_EL AArch64 system registers. These "key registers" are

specific system registers, which are special registers on ARM systems which control various behaviors/controls/statuses for the system, and are responsible for maintaining the current keys used for signing and authenticating pointers. These two registers ("lo" and hi") each hold a single 64-bit value, which results in a concatenated 128-bit key. EL1, in this case, refers to *exception level* "1" - which denotes the ARM-equivalent of "privilege level" the CPU is running in (as ARM-based CPUs are "exception-oritented", meaning system calls, interrupts, etc. are all treated as "exceptions"). Typically EL1 is associated with kernel-mode. User-mode and kernel-mode, for Windows, share EL1's signing key register (although the "current" signing key in the register changes depending on if a processor is executing in kernel-mode or user-mode). It should be noted that although the signing key for user-mode is stored in an EL1 register, the register itself (e.g., reading/writing) is *inaccessible* from user-mode (EL 0).

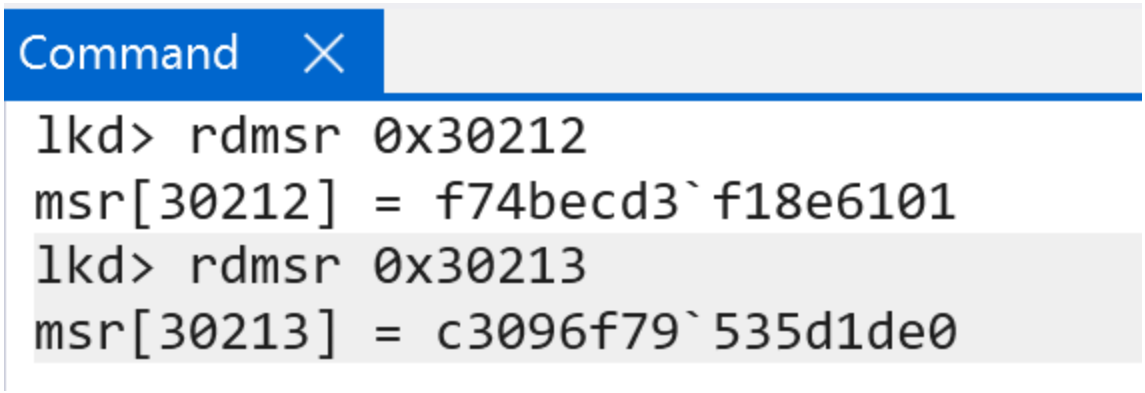
```
if ( !prcb->Number )
{
  loaderParameterExtension = LoaderParameterBlock->Extension;
  g_HypervisorVendorIdentity = loaderParameterExtension->HypervisorIdentity;

  //
  // PointerAuthKernelIpEnabled = bit 16
  //
  if ( (loaderParameterExtension->LoaderParameterFlags & 0x10000) != 0
    && HviIsAnyHypervisorPresent()
    && HviIsHypervisorVendorMicrosoft() )
  {
    KePointerAuthEnabled |= 4u;
    KePointerAuthKernelIpKey = *LoaderParameterBlock->Extension->PointerAuthKernelIpKey;
  }
}
if ( (KePointerAuthEnabled & 4) != 0 && !prcb->Number )
{
  //
  // Bit 30 = Enables pointer auth of instruction addresses using APIBKey_EL1
  //
  _WriteStatusReg(SCTLR_EL1, _ReadStatusReg(SCTLR_EL1) | 0x40000000);
  _WriteStatusReg(APIBKeyHi_EL1, KePointerAuthKernelIpKey.High);
  _WriteStatusReg(APIBKeyLo_EL1, KePointerAuthKernelIpKey.Low);
  __isb(_ARM64_BARRIER_SY);
}
```

It is possible to examine the current signing key values using WinDbg. Although WinDbg, on ARM systems, has no extension to read from these system registers, it was discovered through trial-and-error that it is possible to leverage the rdmsr command in WinDbg to read from ARM system registers using the encoding values provided by the ARM documentation. The two PAC key system registers used by Windows have the following encodings:

```
1. APIBKeyLo_EL1 - op0: 0b11 (3) - op1: 0b000 (0) - CRn: 0b0010 (2) - CRm:
0b0001 (1) - op2: 0b010 (2)
```

2. APIBKeyHigh_EL1 - op0: 0b11 (3) - op1: 0b000 (0) - CRn: 0b0010 (2) - CRm: 0b0001 (1) - op2: 0b011 (3)

Concatenating these binary values into their hexadecimal values, it is then possible to leverage the rdmsr command to view the current signing key values:



After the initial signing key value has been configured, the kernel continues executing its entry point in order to continue to fill out some of the basic functionality of the kernel (although the kernel is not done yet being fully initialized). Almost immediately after performing basic PAC initialization, the function KiInitializeBootStructures is called from the kernel entry point, which *also* receives the loader parameter block and initializes various items such as the feature settings bitmask, setting the proper stack sizes (especially for "special" stacks like ISR stacks and DPC stacks), etc. One of those crucial things that this function does is call into KiDetectPointerAuthSupport, which is responsible for the bulk of the PAC initialization. This function is responsible for reading from the appropriate PAC-related ARM system registers in order to determine what specific PAC features the current CPU is capable of supporting.

```
//
// APA (bits 7:4)
//
apaBits = _ReadStatusReg(ID_AA64ISAR1_EL1) >> 4;
if ( apaBits )
{
  //
  // apaBits are at least 1. apaBits < 4 = QARMA5 algorithm is supported.
  // Anything "> 4" means FEAT_FPAC is available (faulting on AUT* instruction is supported)
  //
  if ( apaBits < 4 )
    pointerAuthFeaturesSupported = 3;
  else
    pointerAuthFeaturesSupported = 0x13;
}

//
// API (bits 11:8)
//
apiBits = (_ReadStatusReg(ID_AA64ISAR1_EL1) >> 8) & 0xF;
if ( apiBits )
{
  if ( (pointerAuthFeaturesSupported & 1) != 0 )
    KeBugCheck2(0x5Du, 9, 0, 0, 0, 0);
  addressAuthSupported = pointerAuthFeaturesSupported | 1;
  pointerAuthFeaturesSupported = pointerAuthFeaturesSupported | 1u;
  if ( apiBits >= 4 )
    //
    // FEAT_FPAC is the minimum implemented address auth algorithm
    //
    pointerAuthFeaturesSupported = addressAuthSupported | 0x10u;
}

//
// GPA (bits 27:24)
//
if ( (_ReadStatusReg(ID_AA64ISAR1_EL1) & 0xF000000) != 0 )
  //
  // Generic authentication is supported via QARMA5 (includes PACGA instruction)
  //
  pointerAuthFeaturesSupported = pointerAuthFeaturesSupported | 0xCu;
if ( _ReadStatusReg(ID_AA64ISAR1_EL1) >> 28 )
{
  if ( (pointerAuthFeaturesSupported & 4) != 0 )
    KeBugCheck2(0x5Du, 10, 0, 0, 0, 0);
  //
  // FEAT_PACIMP is implemented
  //
  pointerAuthFeaturesSupported = pointerAuthFeaturesSupported | 4u;
}
if ( ProcessorNumber )
{
  if ( pointerAuthFeaturesSupported != KePointerAuthSupported )
    KeBugCheck2(0x3Eu, ProcessorNumber, pointerAuthFeaturesSupported, KePointerAuthSupported, 0, 0);
}
else
{
  KePointerAuthSupported = pointerAuthFeaturesSupported;
}
```

After the current CPU's supported options are configured, "phase 0" of the system initialization process (achieved via KeInitsystem) will fully enable PAC. Currently, on Windows 11 24H2 and 25H2 preview builds, enablement is gated through a feature flag called Feature_Pointer_Auth_User__private_featureState. If the feature flag is enabled, a secondary check is performed to determine if a registry override option to disable PAC was present. Additionally, if the PAC feature flag is disabled, a check is performed to see if a registry override to enable PAC is present. The applicable registry paths are:

- HKLM\System\CurrentControlSet\Control\Session Manager\Kernel\PointerAuthUserIpEnabled

- HKLM\System\CurrentControlSet\Control\Session Manager\Kernel\PointerAuthUserIpForceDisabled

```
  switch ( Phase )
  {
    case 0:
      pcr = KeGetPcr();
      KiRcuSystemInitialize();
      KeInitializeSchedulerAssist(&pcr->Prcb);

      //
      // 0x567265707948734D == Microsoft hypervisor identity
      //
      if ( (KePointerAuthSupported & 1) != 0 && g_HypervisorVendorIdentity == 0x567265707948734DLL )
      {
        if ( Feature_Pointer_Auth_User__private_IsEnabledDeviceUsageNoInline() )
        {
          if ( !KePointerAuthUserIpForceDisabledFromReg )
EnablePointerAuthViaReg:
            KePointerAuthEnabled |= 1u;
        }
        else if ( KePointerAuthUserIpEnabledFromReg )
        {
          goto EnablePointerAuthViaReg;
        }
      }
```

Note that the "enablement" flags are not directly tied one-to-one to the "supported flags". As previously seen, KePointerAuthEnabled is masked with the value 4 in KiSystemStartup before the "supported" options are even evaluated. Additionally, note that the KePointerAuthEnabled variable is marked as read-only and is present in the CFGRO section, which is *also* read-only in the VTL 0 guest page tables (known in ARM as the “Stage 2 tables” with “Stage 2” tables being the final level of translation from guest memory to system memory) thanks to the services of Hypervisor-Protected Code Integrity (HVCI), along with KePointerAuthKernelIpKey and KePointerAuthMask. As seen below, even using WinDbg, it is impossible to overwrite these global variables as they are read-only in the “Stage 2” page tables.

```
Command
lkd> !pte nt!KePointerAuthEnabled
                                           VA fffff802b1401880
PXE at FFFF9D4EA753AF80    PPE at FFFF9D4EA75F0050    PDE at FFFF9D4EBE00AC50    PTE at FFFF9D7C0158A008
contains 0060000081713F23  contains 0060000081712F23  contains A060000881000781  contains 0000000000000000
pfn 81713      -R--ADK--V  pfn 81712      -R--ADK--V  pfn 881000     -R-GA-K-LV  LARGE PAGE pfn 881001

lkd> dd nt!KePointerAuthEnabled L1
fffff802`b1401880  00000005
lkd> ep nt!KePointerAuthEnabled 0
                               ^ Memory access error in 'ep nt!KePointerAuthEnabled 0'
```

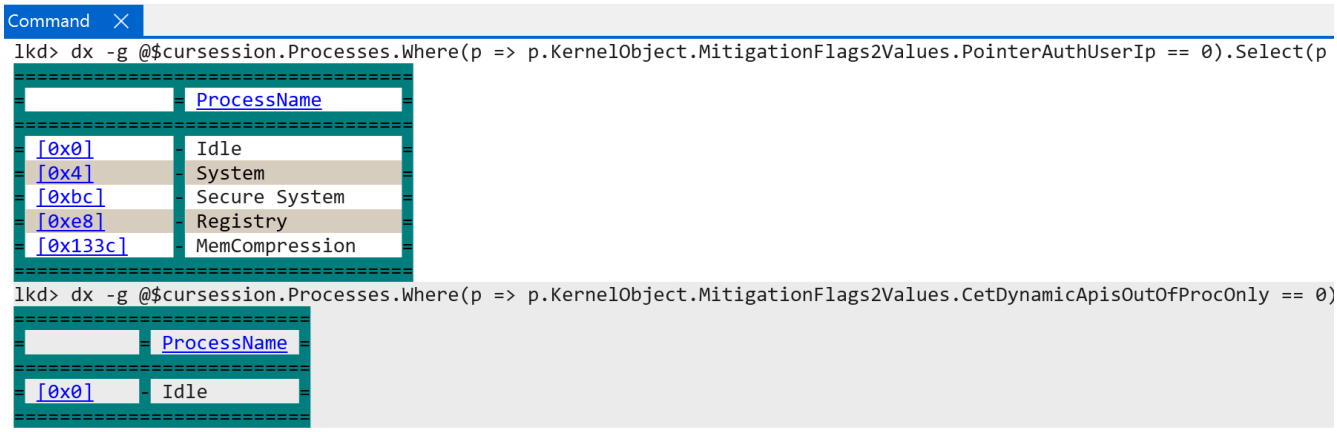
As an aside, the supported *and* enabled PAC features can be queried via NtQuerySystemInformation through the SystemPointerAuthInformation class.

```
C:\>C:\WindowsPAC.exe [+] System Pointer Authentication Control (PAC)
settings: [>] SupportedFlags: 0x1F [>] EnabledFlags: 0x101 [*]
AddressAuthFaulting: TRUE [*] AddressAuthQarma: TRUE [*] AddressAuthSupported:
TRUE [*] GenericAuthQarma: TRUE [*] GenericAuthSupported: TRUE [*]
KernelIpAuthEnabled: TRUE [*] UserGlobalIpAuthEnabled: FALSE [*]
UserPerProcessIpAuthEnabled: TRUE
```

Once the appropriate PAC-related initialization flags have been set, PAC is then enabled on a per-process basis (if per-process PAC is supported, which currently on Windows it is). For user-mode PAC, the enablement process begins at process creation, specifically during the allocation of the new process object. If PAC is enabled, each user-mode process (meaning EPROCESS->Flags3.SystemProcess is not set) is unconditionally opted-in to PAC (as all kernel-mode code shares a global signing key).

```
  //
  // Unconditionally set PAC to on
  //
  atomic_fetch_or(&TargetProcessObject->MitigationFlags2Values, 0x8000000u);
EnablePointerAuthAuditMode:
  if ( ((mitigationAuditOptions >> 12) & 3) == 1 )
    atomic_fetch_or(&TargetProcessObject->MitigationFlags2Values, 0x10000000u);
  redirectionTrustPolicy = WORD1(mitigationOptions) & 3;
  if ( redirectionTrustPolicy )
  {
    if ( redirectionTrustPolicy != 1 )
      goto SetRedirectionTrustPolicy;
  }
  else if ( (Flags & 0x80000) != 0 )
  {
    goto SetRedirectionTrustPolicy;
  }
  //
  // Unconditionally set CetDynamicApisOutOfProcOnly
  //
  atomic_fetch_or(&TargetProcessObject->MitigationFlags2Values, 0x40000000u);
```

Additionally, likely as a side effect of Intel CET enablement on x86-based installations of Windows, the mitigation value CetDynamicApisOutOfProcOnly is also set unconditionally for *every* process except for the Idle process on Windows.



For the sake of completeness, the CET dynamic address range feature is not actually supported as the PROCESSINFOCLASS enum value ProcessDynamicEnforcedCetCompatibleRanges, for the NtSetInformationProcess system service, always returns STATUS_NOT_SUPPORTED on Windows ARM systems.

```
case ProcessFiberShadowStackAllocation:
case ProcessFreeFiberShadowStackAllocation:
case ProcessAltSystemCallInformation:
case ProcessDynamicEHContinuationTargets:
case ProcessDynamicEnforcedCetCompatibleRanges:
case ProcessEnableOptionalXStateFeatures:
  return STATUS_NOT_SUPPORTED;
```

Returning to user-mode PAC, Windows SDK contains two documented ways to enable/disable PAC for user-mode processes. For extended process creation parameters, the following parameters are available in the SDK:

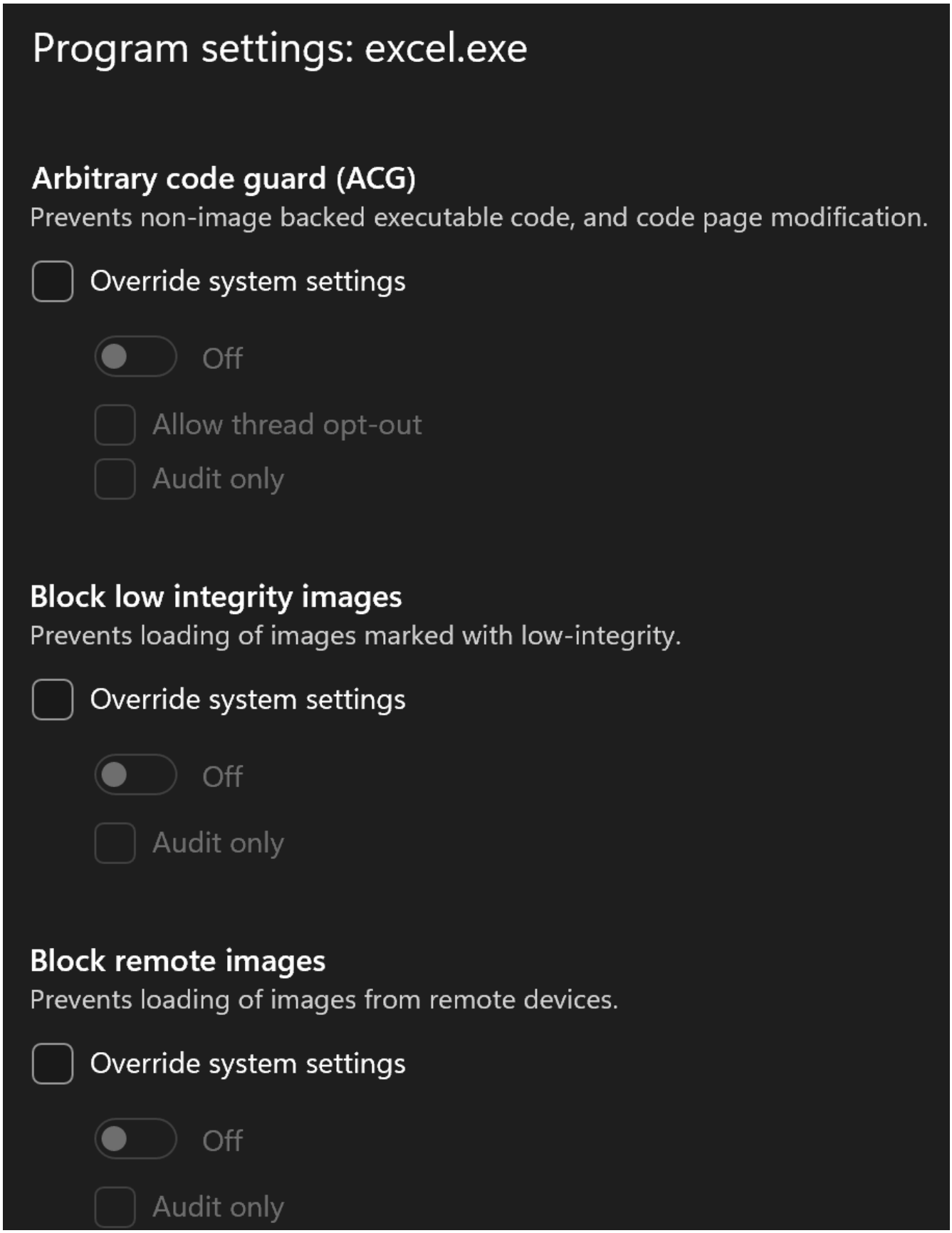
```
//
// Define the ARM64 user-mode per-process instruction pointer authentication
// mitigation policy options.
//



#define PROCESS_CREATION_MITIGATION_POLICY2_POINTER_AUTH_USER_IP_MASK      (0x00000003u

#define PROCESS_CREATION_MITIGATION_POLICY2_POINTER_AUTH_USER_IP_DEFER     (0x00000000u

#define PROCESS_CREATION_MITIGATION_POLICY2_POINTER_AUTH_USER_IP_ALWAYS_ON (0x00000001u

#define PROCESS_CREATION_MITIGATION_POLICY2_POINTER_AUTH_USER_IP_ALWAYS_OFF
(0x00000002ui64 << 44)

#define PROCESS_CREATION_MITIGATION_POLICY2_POINTER_AUTH_USER_IP_RESERVED  (0x00000003u
```

Additionally, for runtime enablement/disablement, the following structure can be supplied with the ProcessUserPointerAuthPolicy:

```
typedef struct _PROCESS_MITIGATION_USER_POINTER_AUTH_POLICY { union { DWORD
Flags struct { DWORD EnablePointerAuthUserIp : 1; DWORD ReservedFlags : 31; }
DUMMYSTRUCTNAME; } DUMMYUNIONNAME; }
PROCESS_MITIGATION_USER_POINTER_AUTH_POLICY,
*PPROCESS_MITIGATION_USER_POINTER_AUTH_POLICY;
```

However, testing and reverse engineering revealed that PAC is unconditionally enabled on user-mode processes (as shown above) with no way to disable the mitigation either at process creation (e.g., creating a child process with extended parameters) or by calling

SetProcessMitigationPolicy at runtime. The only other supported way to enable a process mitigation at process creation is to use the ImageFileExecutionOptions (IFEO) registry key. This functionality is wrapped by the "Exploit Protection" UI on Windows systems, but the registry value can be set manually. Unfortunately, there is no PAC Exploit Protection setting in the UI.



Outside of the exploit mitigation policy for PAC, there is also an *audit-mode* exploit mitigation policy option in the ImageFileExecutionOptions policy map. This can be confirmed through the

presence of the mitigation flag values of AuditPointerAuthUserIp and AuditPointerAuthUserIpLogged in the MitigationFlags2Values field of a process object on Windows.

The IFEO registry key contains a list of processes that have IFEO values. One of the items encapsulated in the IFEO key, as mentioned, is both the mitigation policy settings and *audit-mode* mitigation policy settings (meaning that an ETW event is logged but the target operation is not blocked/process is not terminated by a mitigation violation) for a target process. These per-process mitigation values are used in making considerations about what mitigation policies will be applied to a particular target process at process creation time. On a default installation of Windows 11 24H2 running an ARM build of Windows, no processes have the audit-mode PAC flags set.

```
Command
lkd> dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.AuditPointerAuthUserIp != 0).Count(),d
@$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.AuditPointerAuthUserIp != 0).Count(),d : 0
lkd> dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.AuditPointerAuthUserIpLogged!= 0).Count(),d
@$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.AuditPointerAuthUserIpLogged!= 0).Count(),d : 0
```

Further investigation reveals that this is because there is no way to set the PAC audit-mode exploit policy value on a per-process basis, even through the IFEO key. This is because if pointer authentication is enabled, for example, the slot in the map (represented by the 0x000000000000X000 nibble) in which audit-mode PAC may be enabled is explicitly overridden by PspAllocateProcess (and no ETW event exists in the manifest of the Microsoft-Windows-Security-Mitigations ETW provider for PAC violations).

```
if ( (KePointerAuthEnabled & 1) == 0 || (processFlags & 3) != 0 || (targetEprocess->Flags3 & 0x1000) != 0 )
  mitigationFlagsMap = mitigationFlagsMap & 0xFFFFFFFFFFFFCFFFuLL | 0x2000;
if ( (processFlags & 4) != 0 && (KePointerAuthEnabled & 1) != 0 )
{
  if ( (parentProcess->MitigationFlags2Values.MitigationFlags2 & 0x8000000) != 0 )
    parentPointerAuthFlags = 1;
  else
    parentPointerAuthFlags = 2;
  mitigationFlagsMap = mitigationFlagsMap & 0xFFFFFFFFFFFFCFFFuLL | ((parentPointerAuthFlags & 3) << 12);
```

Once PAC support has been instantiated for the process, the per-process signing key is configured. Yes, this means that each process has its *own* key it can use to sign pointers. This occurs in PspAllocateProcess and, if a process has not opted in to inheriting the signing key, a random key is generated with BCryptGenRandom.

```
//
// Set the user PAC signing key.
// Bit 27 = MitigationFlags2Values.PointerAuthUserIp
//
if ( (*&targetProcessObject->MitigationFlags2Values & 0x8000000) != 0 )
{
  //
  // Inherit the key from the parent (if configured to do so)
  //
  if ( (processFlags & 4) != 0 )
  {
    *targetProcessObject->PointerAuthUserIpKey = *parentProcess->PointerAuthUserIpKey;
  }
  else
  {
    status = BCryptGenRandom(targetProcessObject->PointerAuthUserIpKey, 16);
    if ( (status & 0x80000000) != 0 )
      goto Exit;
  }
}
```

The "per-process" signing key differs from the initial (kernel) signing key that was configured in KiSystemStartup. This is because, obviously, execution is in kernel mode when the initial signing key is instantiated. However, the implementation of PAC on Windows (as we can see above) instruments a *per-process* signing key (along with a single kernel key). When execution transitions into user mode, the signing key system register(s) are updated to the current process signing key (which is maintained through a process object). The example below outlines the current PAC signing key being updated to that of a user-mode process, specifically when a return into user-mode happens after a system call is handled by the kernel (KiSystemServiceExit).

```
if ( KePointerAuthEnabled )
{
  systemControlReg = _ReadStatusReg(SCTLR_EL1);
  currentThread = KeGetCurrentThread()->Process;
  pacPolicy = (currentThread->MitigationFlags2Values.MitigationFlags2 >> 27) & 1;
  instructionPointerPacPolicy = systemControlReg & 0xFFFFFFFFBFFFFFFFuLL | (pacPolicy << 30);
  if ( systemControlReg != instructionPointerPacPolicy )
    _WriteStatusReg(SCTLR_EL1, instructionPointerPacPolicy);
  if ( pacPolicy )
  {
    curProcessPointerAuthKey = &currentThread->PointerAuthUserIpKey;
    hi = curProcessPointerAuthKey->High;
    lo = curProcessPointerAuthKey->Low;
  }
  else
  {
    hi = 0;
    lo = 0;
  }
  _WriteStatusReg(APIBKeyHi_EL1, hi);
  _WriteStatusReg(APIBKeyLo_EL1, lo);
```

This is how the necessary PAC infrastructure is updated for user-to-kernel and kernel-to-user transitions and how kernel-mode and user-mode PAC on Windows is set up. Let's now examine what Windows does when the proper infrastructure is in place.

Windows PAC As An Exploit Mitigation

Windows currently offers an implementation of PAC (with the ability to expand in the future). Windows currently supports PAC for signing and authenticating "instruction pointers". The way that this manifests itself, however, really results in the signing of return addresses. On Windows, for both user-mode and kernel-mode ARM64 code, one can specify the /guard:signret(-) compiler flag to either explicitly enable or disable the signing of return addresses. Enabling this flag instruments the pacibsp and autibsp instructions into the prologue and epilogue of each function, which are "PAC" instructions used to both sign and subsequently validate return addresses.

In the ARM64 architecture, the semantics of preserving return addresses across call boundaries slightly differ from Intel x86. On x86-based systems, a call instruction will also push the target return address onto the stack. Then, right before a return, the aforementioned return address is "popped" off of the stack and loaded into the instruction pointer. On ARM64 the bl (Branch with Link, similar to a call) instruction will instead place the current, in-scope return address an architectural register (lr, or "link register") with a typical operating system, like Windows, also storing this value on the stack to preserve the return address so the lr register can be used for the *next* call's return address (meaning the return addresses are still stored on the stack on ARM, even with the presence of lr).

The pacibsp instruction will use "key b" (APIBKeyLo_EL1 and APIBKeyHi_EL1) and the value of the in-scope stack pointer to sign the return address. The target return address will remain in this state, with the upper bits (non-canonical) being transformed through the signing.

```
Command
0:000> u @pc L1
WindowsPAC!wmain [C:\Users\conno\source\repos\WindowsPAC\WindowsPAC\Main.cpp @ 492]:
00007ff7`e1eaaa18 d503237f pacibsp
0:000> r @lr
lr=00007ff7e1eabc78
0:000> t
WindowsPAC!wmain+0x4:
00007ff7`e1eaaa1c a9bd7bfd stp         fp,lr,[sp,#-0x30]!
0:000> r @lr
lr=197d7ff7e1eabc78
```

This assumes, however, that there is already a return address to process. What if a user-mode thread, for example, is just entering its initial execution, and there is no return address?

Windows has two functions (for user-mode and kernel-mode) that will generate the necessary "first" signed return address via KiGenerateSignedReturnAddressForStartUserThread. These functions accept the initial stack value as the value to use in the signing of the return address, using instead the pacib instruction, which is capable of using a general-purpose architectural register in the signing process instead of just defaulting to "the current stack pointer".

```
KiGenerateSignedReturnAddressForStartUserThread
                                        ; CODE XREF: KiInitializeContextThread+13C↑p
                                        ; DATA XREF: .pdata:000000001400B3850↑o
                MOV             X1, X0
                ADRL            X0, KiStartUserThread
                PACIB           X0, X1
                RET
```

At this point, the return address (stored in lr, but also present on the stack) has been signed. The in-scope function performs its work and eventually the epilogue of a function is reached (which is responsible for returning to the caller for the current function). When the epilogue is reached, but before the ret has been executed, the autibsp instruction is used to *authenticate* the return address (in lr) before performing the return control-flow transfer. This will result in transforming the value in lr back to the "original" return address so that the return occurs back into a valid memory address.

```
Command
0:000> g
Breakpoint 1 hit
WindowsPAC!wmain+0x68:
00007ff7`e1eaaa80 d50323ff autibsp
0:000> r @lr
lr=197d7ff7e1eabc78
0:000> t
WindowsPAC!wmain+0x6c:
00007ff7`e1eaaa84 d65f03c0 ret
0:000> r @lr
lr=00007ff7e1eabc78
0:000> u @lr
WindowsPAC!invoke_main+0x48 [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 90]:
00007ff7`e1eabc78 2a0003e0 mov         w0,w0
00007ff7`e1eabc7c 2a0003e0 mov         w0,w0
00007ff7`e1eabc80 a8c37bfd ldp         fp,lr,[sp],#0x30
00007ff7`e1eabc84 d50323ff autibsp
00007ff7`e1eabc88 d65f03c0 ret
00007ff7`e1eabc8c 00000000 ???
00007ff7`e1eabc90 00000000 ???
00007ff7`e1eabc94 00000000 ???
```

The effectiveness of PAC, however, relies on what happens if a return address has been corrupted with a malicious return address, like a ROP gadget or the corruption of a return address through a stack-based buffer overflow. In the example below, this is outlined by corrupting a return address on the stack with *another* return address on the stack. Both of these addresses used in this memory corruption example are signed, but, as we can recall from earlier, return addresses are signed with the considerations of the current in-scope stack

pointer (meaning they are tied to a stack frame). Because the corrupted return address does not correspond to an "in-scope" stack frame, the authentication of the in-scope return address (which has been corrupted) results in a fastfail with the code FAST_FAIL_POINTER_AUTH_INVALID_RETURN_ADDRESS - and the application crashes. One interesting note, as you can see, is that WinDbg can convert a signed return address on the stack to its *actual* unsigned value (and appropriate symbol name).

```
Command
0:000> k
 # Child-SP          RetAddr               Call Site
00 00000009`238ffa80 00007ff6`0e0fbc78     WindowsPAC!wmain+0x4c [C:\Users\conno\source\repos\WindowsPAC\WindowsPAC\Main.cpp @ 493]
01 00000009`238ffb90 00007ff6`0e0fb8a4     WindowsPAC!invoke_main+0x48 [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 90]
02 00000009`238ffbc0 00007ff6`0e0fb6c4     WindowsPAC!__scrt_common_main_seh+0x18c [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 288]
03 00000009`238ffc00 00007ff6`0e0fbe44     WindowsPAC!__scrt_common_main+0x14 [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 330]
04 00000009`238ffc10 00007ffe`1d148740     WindowsPAC!wmainCRTStartup+0x14 [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_wmain.cpp @ 16]
05 00000009`238ffc30 00007ffe`1eba43b4     KERNEL32!BaseThreadInitThunk+0x40
06 00000009`238ffc40 00000000`00000000     ntdll!RtlUserThreadStart+0x44
0:000> dps @sp+0x118 L1
00000009`238ffb98  29527ff6`0e0fb8a4 WindowsPAC!__scrt_common_main_seh+0x18c [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 288]
0:000> dps @sp+0x148 L1
00000009`238ffbc8  d819fff6`0e0fb6c4 WindowsPAC!__scrt_common_main+0x14 [D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 330]
0:000> ep 00000009`238ffb98 d819fff6`0e0fb6c4
0:000> g
(dd4.232c): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
Subcode: 0x49 FAST_FAIL_POINTER_AUTH_INVALID_RETURN_ADDRESS Pointer authentication (PAC) return address violation
WindowsPAC!invoke_main+0x54:
00007ff6`0e0fbc84 d50323ff autibsp
```

Shifting focus slightly, when a *kernel-mode* PAC violation, identical to the previous scenario, occurs, a KERNEL_SECURITY_CHECK_FAILURE ensues, with the type of memory safety violation being FAST_FAIL_POINTER_AUTH_INVALID_RETURN_ADDRESS.

```
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure.  The corruption
could potentially allow a malicious user to gain control of this machine.
Arguments:
Arg1: 0000000000000049, Type of memory safety violation
Arg2: ffffd486a2208560, Address of the trap frame for the exception that caused the BugCheck
Arg3: ffffd486a2208420, Address of the exception record for the exception that caused the BugCheck
Arg4: 0000000000000000, Reserved
```

Secure Kernel And PAC

The curious reader may notice that the kernel itself is responsible for managing the key values for PAC. Additionally, we already covered the fact that the in-memory variable which tracks the kernel's PAC signing key (used to sign kernel pointers) is read-only in VTL 0 memory thanks to the services of HVCI. However, the in-memory representation is simply a reflection of the system register value(s) we have talked about before - the APIBKeyLo_EL1 and APIBKeyHi_EL AArch64 registers (specifically when execution is in kernel-mode, loading the per-boot kernel-mode PAC key). What is preventing an attacker, in kernel-mode, from modifying the contents of this system register at any given time? After all, the register is writable from kernel-mode because the configuration is not delegated to a higher security

boundary? To help alleviate this problem, Secure Kernel Patch Guard, more commonly referred to as “HyperGuard” - a security feature promulgated by the Secure Kernel - is used! HyperGuard achieves much of what PatchGuard attempts to defend against (modification of kernel data structures, MSRs on x86 systems, control registers, etc.) but it does so *deterministically*, as opposed to PatchGuard, because HyperGuard runs at a higher security boundary than the code it is attempting to defend (VTL 0’s kernel).

HyperGuard uses what is known as *extents*, which are definitions of what components/code/data/etc. should be protected by HyperGuard. On ARM64 installations of Windows, an ARM64-specific HyperGuard extent exists - the PAC system register extent. This extent is used by HyperGuard to ask the hypervisor to intercept certain items of interest - such as modifications to an MSR (or ARM64 system register), certain memory access operations, etc. Specifically for the ARM64 version of the Secure Kernel, an extent is registered for monitoring modifications to the PAC key system registers. This is done in securekernel!SkpgxInitializeInterceptMasks.

```
//
// Kernel-mode PAC intercept (Secure Kernel)
//
if ( ShvlIsRegisterInterceptAvailable(0x4002F) && (SkpgArm64ReadRegister64(0x40002) & 0x40000000) != 0 )
{
  SkpgArm64PointerAuthKernelEnabled = 1;
  pointerToSigningKey = &SkpgArm64PointerAuthKernelIpKey;
  SkpgArm64PointerAuthKernelIpKey.Low = SkpgArm64ReadRegister64(0x4002E);
  SkpgArm64PointerAuthKernelIpKey.High = SkpgArm64ReadRegister64(0x4002F);
}
```

Although ARM-based hypervisors do not have “Virtual Machine Control Structure”, or VMCS (in the “canonical” sense that x86-based systems do, such as having dedicated instructions in the ISA for reading/writing to the VMCS), ARM hypervisors still must maintain the “state” of a guest. This, obviously, is used in situations like when a processor starts executing in context of the hypervisor software (due to a hypervisor call (HVC call), or other exceptions into the hypervisor), or when a guest starts resuming its execution. Part of this state - as is the case with x86-based systems - is the set of *virtual* registers (e.g., registers which are preserved across exception level changes into/out of the hypervisor and are specific to a guest). Among the virtual registers which are configurable by the hypervisor are, as you may have guessed, the “lo” and “hi” PAC signing key registers! This is what the function from the screenshot above intends to achieve - securekernel!SkpgArm64ReadRegister64.

Microsoft documents many of the 64-bit virtualized-registers. Among the undocumented registers, however, are the ARM-based virtualized registers. However, we can see above that values 0x4002E and 0x4002F correspond to the virtual/private PAC signing registers. For completeness sake, 0x40002 corresponds to SCTLR_EL1. This was determined by

examining the bit being processed (bit 30, via the 0x40000000 mask). This was previously seen, in the beginning of our analysis, by the toggling of SCTLR_EL1.EnIB bit (bit 30).

This entire configuration allows the Secure Kernel to intercept, via HyperGuard, any unauthorized modification of the PAC signing key register.

Conclusion

ARM-based processors, without the presence of backwards-edge control flow integrity (CFI) mitigations like CET, are able to effectively leverage PAC to defend against return address corruption. Windows, as we have seen, currently leverages PAC only in limited circumstances (like the protection of return addresses), which is standard on many mainstream implementations of PAC (with the ability in the future, if feasible, to expand into protection of data accesses). PAC provides a viable solution to protect non-x86-based processors from certain classes of memory corruption exploits. In addition, current-generation ARM64 Microsoft devices, like the Surface Pro, are not shipped with chips that can support the Memory Tagging Extension (MTE) feature. Although not implemented today on Windows systems, the implementation of both PAC and MTE in the future would serve to greatly increase the cost of memory corruption exploits. Given the protections afforded by the hypervisor, plus the current implementation of PAC, ARM-based Windows provides both user-mode and kernel-mode code with additional security against memory corruption exploits.