CET Updates – Dynamic Address Ranges

windows-internals.com/cet-updates-dynamic-address-ranges

By Yarden Shafir

In the last post I covered one new addition to CET – relaxed mode. But as we saw, there were a few other interesting additions. One of them is `CetDynamicApisOutOfProcOnly`, which is the one I will be covering in this post and which was also backported to `20H1` and `20H2`.

But before I explain the flag, let's talk about the mechanism that it mitigates.

Dynamic Enforced Address Ranges

As we know, Microsoft's implementation of hardware CET prevents a process from setting the instruction pointer to non-approved values through backward edge ("return") flows, including through OS-provided mechanisms. Whether it's by returning to an address that's different from the one that's in the shadow stack, or setting the thread context, or unwinding to an unexpected address during exception handling. But like we've seen in the last two posts, there are cases that require special handling. One of those is dynamically generated (JIT) code.

Such code doesn't always follow the rules and assumptions of CET, so Microsoft added a way to handle its needs, similar to the handling of Dynamic Exception Handler Continuation Targets, which I talked about in the first post. In this solution, a process can declare some ranges as "CET compatible" such that setting the instruction pointer to any address within that range won't trigger a CET exception (`#CP`) that will crash the process.

To keep those ranges, the `EPROCESS` received a new field:

```
typedef struct _EPROCESS
{
    ...
    /* 0x0b18 */ struct _RTL_AVL_TREE DynamicEHContinuationTargetsTree;
    /* 0x0b20 */ struct _EX_PUSH_LOCK DynamicEHContinuationTargetsLock;
    /* 0x0b28 */ struct _PS_DYNAMIC_ENFORCED_ADDRESS_RANGES DynamicEnforcedCetCompatibleRanges;
    /* 0x0b38 */ unsigned long DisabledComponentFlags;
    ...
} EPROCESS, *PEPROCESS;
```

This new `PS_DYNAMIC_ENFORCED_ADDRESS_RANGES` structure contains an `RTL_AVL_TREE` and an `EX_PUSH_LOCK`. New ranges are inserted into the tree through a call to `NtSetInformationProcess` with the new information class `ProcessDynamicEnforcedCetCompatibleRanges` (`0x66`). The caller supplies a pointer to a `PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE_INFORMATION` structure as the `ProcessInformation` argument, which contains the ranges to insert into the tree, or remove from it, depending on the Flags field:

```
typedef struct _PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE
{
    ULONG_PTR BaseAddress;
    SIZE_T Size;
    DWORD Flags;
} PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE, *PPROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE;

typedef struct _PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGES_INFORMATION
{
    WORD NumberOfRanges;
    WORD Reserved;
    DWORD Reserved2;
    PPROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE Ranges;
} PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGES_INFORMATION, *PPROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGES_INFORMATION;
```

The ranges are then read from the structure and inserted into the tree by the `PspProcessDynamicEnforcedAddressRanges` function. Of course, the process doesn't have to call `NtSetInformationProcess` directly, as there is a wrapper function for this in the `Win32` API exposed by `KernelBase.dll` – `SetProcessDynamicEnforcedCetCompatibleRanges`:

```
BOOL
SetProcessDynamicEnforcedCetCompatibleRanges (
    _In_ HANDLE ProcessHandle,
    _In_ WORD NumberOfRanges,
    _In_ PPROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE Ranges
    )
{
    NTSTATUS status;
    PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGES_INFORMATION dynamicEnforcedAddressRanges;
    dynamicEnforcedAddressRanges.NumberOfRanges = NumberOfRanges;
    dynamicEnforcedAddressRanges.Ranges = Ranges;
    status = NtSetInformationProcess(ProcessHandle,
                                     ProcessDynamicEnforcedCetCompatibleRanges,
                                     &dynamicEnforcedAddressRanges,
                                     sizeof(PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGES_INFORMATION));
    if (NT_SUCCESS(status))
    {
        return TRUE;
    }
    BaseSetLastNTError(status);
    return FALSE;
}
```

This tree is used every time a CET fault happens – `KiControlProtectionFaultShadow` is invoked. It calls into `KiControlProtectionFault`, which calls `KiProcessControlProtection`. This function will look for the target address in the shadow stack and if it fails, will try the dynamic enforced CET compatible ranges through an exception handler.

First, the handler checks in strict CET is enabled in the system, to know whether it should check if the process has CET enabled (as a reminder, strict CET means that CET checks will be performed on all processes, regardless of how they were compiled). If strict mode is not enabled, the function will check the image headers for the `CETCOMPAT` flags and will skip the ranges check if the flag is not set.

If it was determined that CET should be enforced for the image, the function will call `RtlFindDynamicEnforcedAddressInRanges` to check if the target address is inside one dynamically enforced CET compatible address ranges. The function returns a `BOOLEAN` value to indicate whether a suitable range for the address was found or not. If a range was found, or if for some other reason the process should not be crashed (process is not CET compatible or audit mode is enabled), the function will then call `KiFixupControlProtectionUserModeReturnMismatch` to insert the target address into the shadow stack to allow the process to continue normal execution.

The Mitigation

Looking at all of this, an obvious flaw comes to mind. If a process can declare ranges that will be ignored by CET, all an exploit needs to do to bypass CET is manage to add a useful range in the process memory to the tree, and then ROP itself in the approved range.

This is why the `CetDynamicApisOutOfProcOnly` flag was added – it only allows a process to add dynamic CET compatible ranges for remote processes, and not for themselves. It does a very simple thing – inside `NtSetInformationProcess`, before calling `PspProcessDynamicEnforcedAddressRanges`, the function checks if `CetDynamicApisOutOfProcOnly` is set for the process and if the process is trying to add dynamic CET compatible ranges for itself. If so, the function will return `STATUS_ACCESS_DENIED` and the attempt will fail.

And actually in the newest builds of Windows, almost all Windows processes have this flag set by default. The only process that doesn't appear to have it enabled is the `Idle` process (which doesn't have a real `EPROCESS` structure, only a `KPROCESS`, so we're effectively reading garbage memory).

```
0: kd> dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.CetDynamicApisOutOfProcOnly != 0).Count()
@$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.CetDynamicApisOutOfProcOnly != 0).Count() : 0x8d
0: kd> dx @$cursession.Processes.Count()
@$cursession.Processes.Count() : 0x8e
0: kd> dx @$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.CetDynamicApisOutOfProcOnly == 0)
@$cursession.Processes.Where(p => p.KernelObject.MitigationFlags2Values.CetDynamicApisOutOfProcOnly == 0)
    [0x0]            : Idle [Switch To]
```

Read our other blog posts: