Evaluating the feasibility of enabling SMAP for the Windows kernel

Joe Bialek, Saar Amar – Microsoft Security Response Center (MSRC)

July 1, 2020

Introduction

This paper outlines research that has gone in to utilizing Intel's Supervisor Mode Access Prevention (or SMAP) feature in Windows.

SMAP is a feature that enforces that user-mode accessible virtual addresses cannot be read or written from code running at CPL0 (i.e. kernel-mode code). This allows us to both:

1. Help enforce a safer programming model for kernel-mode code.
2. Make certain user-mode to kernel-mode elevation of privileged vulnerabilities harder to exploit.

Unfortunately, we were not able to find a way to use SMAP in Windows that is both backwards compatible with existing kernel-mode code and performant. We were able to prototype safer kernel programming patterns and found that in many cases these patterns are easier to use and safer in kernel-mode code even without the use of the SMAP feature.

Key Findings

Before diving in, let's view the key findings we had during this research:

1. SMAP is not practical to enable in the Windows kernel. Because SMAP is a breaking change for drivers, the only way we could possibly enable SMAP is on a per-component basis. The performance impact of the instrumentation required to do this makes this approach non-viable.
2. For kernels that do not need to interoperate with existing drivers, such as Secure Kernel, it would be possible for us to enable SMAP as we would not need the instrumentation scheme to enable SMAP on a per-component basis.
3. While SMAP is not viable for the Windows kernel, it is viable in most cases to move Windows kernel code away from the manual probe + copy model and to a model where the probe and copy is done inside a single accessor function.

Future Directions

Even though it does not appear to be practical to enable SMAP for the Windows kernel, there are still some ideas we can consider for future work.

Write New Code With User-Mode Accessor Functions

Even without SMAP support in Windows, we believe that using accessor functions will simplify code and lead to fewer bugs.

We expect that this programming model would make the following vulnerability classes less likely to occur:

1. Double fetch vulnerabilities where data is fetched twice from user-mode and changes in between the two accesses.

2. Issues related to the probe size differing from the copy size, since developers would no longer be calling RtlCopyMemory and the probe functions separately.

Additionally, this programming model tends to make code easier to read. The current programming model can lead to probes being so separated from memory accesses that it's difficult to determine if the correct probe has happened. This can make it confusing to understand if some memory being accessed is in user-mode or kernel-mode. With a user-mode accessor model, all memory being operated on is kernel-mode. User-mode memory is only operated on using the accessor functions.

Enable SMAP For Kernels With No 3rd Party Code Such as Secure Kernel

Currently SMAP is not enabled for Secure Kernel, but since Secure Kernel does not currently have a legacy driver model, we can explore enabling SMAP for VTL1. It's even better, as we don't need an instrumentation here. 100% of the code running in VTL1 kernel-mode is controlled by Microsoft (it's not open for 3rd parties), and since it's relatively small, we can rewrite it and convert it to the Copyin/Copyout model.

Kernel-Mode Programming Primer

In the Windows NT kernel, the virtual address space is split in half (note: on x86 it is possible to split it slightly differently but we'll ignore that and focus on 64-bit x86-64 kernels). Most CPU's currently support 48-bit virtual addressing which provides a total of 256 TB of virtual address space. This means that starting from address 0, user-mode gets 128TB of virtual address space. Starting from address 0xFFFF800'00000000, the kernel gets 128TB of virtual address space.

The virtual address space is controlled by the page tables. Each process has its own top-level page table, managed by the kernel, that controls both the kernel and user-mode virtual address space. Most mappings in the kernel-mode address space are the same for all processes. Some mappings are process dependent, such as session space.

When code running in user-mode makes a system call, execution is transferred to the kernel to handle the system call and the same top-level page table is used (note: not always the case due to some speculative execution mitigations). When running code in user-mode, the user-mode code cannot access kernel-mode virtual address space due to the permission bits in the page table entries. However, the kernel-mode address space is mapped by the page tables. When execution transfers to kernel-mode, the kernel can access all of the kernel-mode virtual address space as well as the user-mode virtual address space.

One convenient side effect of this is that when a system call is made, user-mode can pass the kernel pointers to data that are mapped in the user-mode virtual address space and the kernel-mode code can access those pointers. Note that the kernel doesn't actually trust user-mode processes so it cannot blindly follow pointers that user-mode gives it. If the kernel blindly trusted pointers, user-mode code could tell the kernel "write the results of the operation to address FOO", where address FOO is actually a pointer to a critical kernel-mode data structure. To mitigate this, the kernel must always ensure that the pointers it receives from user-mode code point to the user-mode virtual address space using functions such as ProbeForRead and ProbeForWrite.

Issues With the Current Kernel-Mode Model

The current design of how kernel-mode interacts with user-mode poses a few issues from a security point of view:

1. **Developers must always remember to probe pointers (to ensure they point at a user-mode virtual address) before following them. However, there is no solid way to determine if a developer forgets to do this.** These mistakes can persist for years because as long as user-mode code doesn't maliciously pass a kernel-mode pointer, no problems arise. However, attackers can find these issues and exploit them.
2. **Probe functions return success when the size passed to them is zero.** This can be problematic when combined with integer overflows. A developer may sum up some sizes, probe the resulting size (which is zero, so the probe succeeds), and then do individual copies of data with the smaller sizes that were previously summed.
3. **Developers must be extremely careful not to re-fetch data from user-mode after it has been validated. There are no good ways to automate detecting when developers do this.** Imagine a developer reads a parameter from user-mode and then validates that parameter. Then the developer re-reads the parameter from user-mode and uses it for some critical operation. The data could have changed in between the first time the data was read validated and when it was read the second time.
4. **Having user-mode memory always read/writeable by kernel-mode code makes it easier to exploit certain types of kernel-mode vulnerabilities**. As an example, imagine a vulnerability where an attacker can overflow a buffer and corrupt a pointer to structure FOO immediately after the buffer. Imagine structure FOO contains function pointers. If an attacker can overwrite the pointer to FOO with any arbitrary value, and user-mode virtual address space is readable by kernel-mode code at all times, the attack can overwrite the pointer with a user-mode pointer. When the kernel follows this pointer, it will be reading from a structure that is expected to be controlled by the kernel but is currently mapped in user-mode.

Safer Kernel Programming Model

The issues with the current programming model can roughly be summarized as:

1. Probing of pointers is separate from accessing memory which allows unintended consequences
    a. Probe is missing, memory accesses still work
    b. Size passed to probe is not necessarily the same as size used to access memory
2. It is very easy to forget you are working with shared memory and double-fetch things
3. Having the entire user-mode virtual address space accessible even when executing code that doesn't need this access violates least-privilege principles and makes attackers lives easier

We investigated a safer programming model that entails:

1. The probe and memory access are always paired under a single user-mode accessor API so that developers cannot accidentally access memory without probing the pointer. In other words, you cannot directly call RtlCopyMemory with user-mode addresses, you must go through a new API that explicitly probes the address and then does the copy for you.

2. The user-mode virtual address space in inaccessible to kernel-mode unless inside of one of these user-mode accessor functions. This can be accomplished by using the SMAP feature. This forces developers to use these new accessors; their code will not work if they don't use the accessor.

For this to happen, we want to define new Rtl functions that will do Copyin/Copyout. Those functions will wrap each access to user-mode virtual addresses from kernel-mode code by implementing the following pattern:

a. Call probe, check the integrity of the pointer
b. Enable-user-access
c. Do user access
d. Disable-user-access

Under this model, the expected work flow for a developer would be:

a. User-mode passes a pointer to a structure to a system call
b. Kernel-mode calls RtlCopyStructureFromUser, gets a copy of the structure in the kernel address space
c. Kernel operates on its private copy of the structure (both reads and writes)
d. Before returning to user-mode, the kernel copies the structure back to user space, by calling RtlCopyStructureToUser (if the structure needs to be updated)

SMAP in Windows

Considerations for Enabling SMAP

Enabling SMAP in Windows is a lot more complicated than flipping a switch. Notably, enabling SMAP is a breaking change. By default, user-mode memory will not be accessible from kernel-mode code. Any existing code that interacts with user-mode memory will no longer work.

It's unrealistic to expect all of the drivers in the world to be reimplemented to conform with the new model so we need a design that will let us both gradually shift our components to the new model, while not breaking other kernel-mode drivers.

We also need to ensure that runtime performance is not regressed and code size is not substantially affected.

Existing SMAP in Windows

Windows already supports SMAP in very limited circumstances. The primary scenario it is used is when executing DPC's that have an IRQL of DISPATCH_LEVEL since user-mode access should be illegal. This support was added in 19H1, but of course it doesn't address the goals mentioned above.

The plan – High Level

How to {Enable,Disable}-user-access

We have 2 possible interfaces to change the current mode in the processor for allowing/disallowing user-access, and we need to consider the advantages/disadvantages of each one (for performance and code-gen size). The possibilities are as follows:

- Raw writes to CR4

- stac / clac instructions (sets/clears the AC flag bit in EFLAGS register)

The first one is not an option for us, as for the following reasons:

- There aren't dedicated instructions for writes to CR4.SMAP, which will result in increasing (significantly) the binary size, which will decrease performance significantly
- Writes to CR4 are serialized, which will be a heavy memory barrier

The second option is way better:

- stac / clac instructions are 3 bytes each, so binary size won't increase "too much"
- Might be better than writing to CR4, with respect to memory barrier

Making NTOS SMAP Compatible

All existing Windows kernel code (both 1st and 3rd party) is built under the assumption that it can freely access user-mode memory. Accordingly, there is no way we can enable SMAP (and therefore disable user-mode access) for the world.

The only solution that seems tractable is to convert subsets of code to assume that user-mode access is disabled by default. This code will be forced to go through specific "user-mode accessor" functions when it needs to access user-mode memory.

From a high level, the strategy works as follows:

1. NTOS will enable the SMAP feature via the CR4 register, however, it will keep user-mode access enabled by setting the EFLAGS AC bit to 0. This ensures all current code continues to work.
2. Code that opts-in to the new model of "user-mode disabled by default" will compile with a specific compiler flag.
    a. The compiler creates a list of functions that are a potential "entry-point from code that is running with user-mode access enabled", such as an exported function. The compiler will generate code to disable user-access by setting EFLAGS.AC to 0 in the functions prolog.
    b. Prior to any location in code that could lead to executing code that doesn't support running with user-mode access disabled, such as making a function call through the import address table, the compiler will generate code to enable user-access by setting EFLAGS.AC to 1.

The idea behind this instrumentation is that the compiler ensures that user-mode access gets disabled prior to executing any code in our current binary and that user-mode access gets enabled whenever code execution leaves the current binary. This allows to opt-in specific binaries while still allowing legacy code to function exactly as it does today.

Any time kernel code that uses SMAP needs to access user-mode memory, it must do so through a user-mode accessor function exported from the kernel that will temporarily enable user-mode access, do the access, and then re-disable user-mode access.

Compiler Instrumentation Details

To start with, the compiler needs to generate a list of functions that are potential entry-points into the binary:

1. Address taken functions (could be called indirectly from another binary)
2. Exported functions (could be called from another binary)
3. LTCG (Link Time Code Generation) Specific: Any function that is referenced outside of the LTCG module (for example, by a static libraries or by and object file that is directly linked) because we don't know if that function was compiled with SMAP support (and must assume it doesn't support SMAP to be safe)
4. Non-LTCG Specific: Any function that is called by code from outside the current file because we don't know if that external function was compiled with SMAP support (and must assume it doesn't support SMAP to be safe)

For any function that is an entry point, we do the following entry-point specific instrumentation. The purpose of this is to ensure that user-mode access is disabled by default for the current binary and enabled when returning to code that may not supported running with user-mode access disabled.

- Prolog: Disable user-mode access (clear EFLAGS.AC bit).
- Epilog: Enable user-mode access (set EFLAGS.AC bit).

For all functions (both entry-point functions and functions that are not potential entry-points) we search for both calls and indirect calls and do the following instrumentation.

1. Direct Calls:
    a. If the direct call is to code that may not support running with user-mode access disabled (such as a direct call to a function from a static library), the compiler must enable user-mode access prior to making the call and disable user-mode access after the call returns.
    b. If the direct call is to the CFG check function (which will leave the binary), the compiler must disable user-mode access after the call returns. The CFG check function itself is modified to enable user-mode access to save on binary size. Note that the CFG check function does a tail-call so it is not possible to also disable user-mode access in the CFG check function after the indirect call is made.
    c. If the direct call is to a function in the binary that does support running with user-mode access disabled, but the function is an entry-point, we know the function will end up enabling user-mode access in its epilog. Accordingly, the compiler needs to disable user-mode access after calling this function.
    d. If the direct call is to a function in the binary that does support running with user-mode access disabled and is NOT an entry point, the function can be called without any instrumentation needed.
2. Indirect calls:
    a. User-mode access must be enabled prior to making the indirect call and disabled after the call returns. In the case of binaries compiled with CFG support, user-mode access is enabled in the CFG check function saving binary space.
3. Create explicit accessor functions that:
    a. Probe
    b. Disable SMAP (if the platform supports SMAP)
    c. Access user-mode memory
    d. Re-enable SMAP (if the platform supports SMAP)

kCFG

Another approach would be to convert indirect branches into direct ones, by using trampolines. Then, we can implement any functionality we would like in the trampoline which will be executed before each indirect branch will occur. Actually, we already have this mechanism in NTOS – it's exactly what happens in kCFG. The way kCFG works in NTOS, is that when the flow needs to take an indirect branch, it calls to a trampoline which validates the integrity of the target address (see *_guard_dispatch_icall*). In theory, we could take advantage of this trampoline, and do the allow-user-access before the jump and disable-user-access when we get back. The problem is that we don't use the *call* instruction there, we use *jmp*.

So, we could allow-user-access before the jmp, but it's not trivial how to disable-user-access when we return to NTOS after jump to some non-SMAP compatible kernel module. Please note that it's not trivial to replace this jmp instruction into a call, as it breaks the stack layout with respect to number of arguments passed to the target function. i.e., if a function f() calls a function g(), and pass 6 arguments to it, the first 4 arguments will be set in rcx, rdx, r8, r9, and the last 2 on the stack. But if the kCFG trampoline will execute a call, it will push another return address on the stack, while g() expect a certain layout of arguments. This of course can be solved by implementing many kCFG trampolines per different number of arguments, but it gets quite messy.

Kernel Prototyping

Validating Compiler Instrumentation and Finding Code that Accesses User-Mode

We needed to ensure that the compiler was working correctly. Our intention for prototyping was to enable SMAP in the NT kernel and no other components.

We also needed to gather a list of locations in the kernel that accessed user-mode memory so we could better understand the access patterns that were being used and what sort of accessor functions would be needed.

We created an audit mode for SMAP inside the page fault handler that would simply log any SMAP violation's call stack, re-enable user-mode access by changing the EFLAGS.AC bit, and resume execution. This allowed us to verify there were no SMAP violations occurring in binaries aside from NTOS. It also allowed us to find all the locations accessing user-mode memory when booting the system.

One downside with this approach is that re-enabling user-mode access means that further accesses to user-mode memory won't be logged (unless we end up re-disabling user-mode access in the binary, which would happen if certain things happen in the code such as indirect calls).

To work around this, we created a new compiler mode which will forcibly disable user-access prior to any instruction that may access user-mode memory. Under this model a SMAP violation will occur, will be handled by the page fault handler (which will enable user-access), but prior to another instruction executing that may touch user-mode memory we are guaranteed to re-disable user-mode access first. This allows us to find all locations that touch user-mode memory. It turns out that we have around ~2900 locations that touches user-mode during boot, with 2313 unique locations, in 994 unique functions.

```
fffff800`3a076640  fffff800`38fa0cb8 nt!MiValidFault+0x204
fffff800`3a076648  fffff800`38fa71d9 nt!MiUserFault+0x325
fffff800`3a076650  fffff800`38fa793b nt!MmAccessFault+0x17b
```

```
fffff800`3a076658  fffff800`39226d4f nt!KiPageFault+0x3cf
fffff800`3a076660  fffff800`395103e5 nt!RtlGuardIsValidStackPointer+0x59
fffff800`3a076668  fffff800`38eb6a6e nt!KeVerifyContextRecord+0x7e
fffff800`3a076670  fffff800`394db89a nt!PspSetContextThreadInternal+0x1d2
fffff800`3a076678  fffff800`395f1338 nt!WbSetTrapFrame+0x120
fffff800`3a076680  fffff800`395f3f62 nt!WbHeapExecuteCall+0x226
fffff800`3a076688  fffff800`395f0e54 nt!WbDispatchOperation+0x1a0
fffff800`3a076690  fffff800`395b86fb nt!ExpQuerySystemInformation+0x3633
fffff800`3a076698  fffff800`395b4eff nt!NtQuerySystemInformation+0x7f
fffff800`3a0766a0  fffff800`3922b415 nt!KiSystemServiceCopyEnd+0x35
...
fffff800`3a10f770  00000000`00000000
fffff800`3a10f778  00000000`00000000
fffff800`3a10f780  00000000`00000000
fffff800`3a10f788  00000000`00000000
fffff800`3a10f790  00000000`00000000
fffff800`3a10f798  00000000`00000000
fffff800`3a10f7a0  fffff800`38fa0cb8 nt!MiValidFault+0x204
fffff800`3a10f7a8  fffff800`38fa71d9 nt!MiUserFault+0x325
fffff800`3a10f7b0  fffff800`38fa793b nt!MmAccessFault+0x17b
fffff800`3a10f7b8  fffff800`39226d4f nt!KiPageFault+0x3cf
fffff800`3a10f7c0  fffff800`392fcde6 nt!NtReadFile+0x616
fffff800`3a10f7c8  fffff800`3922b415 nt!KiSystemServiceCopyEnd+0x35
...
fffff800`3a10f770  00000000`00000000
fffff800`3a10f778  00000000`00000000
fffff800`3a10f780  00000000`00000000
fffff800`3a10f788  00000000`00000000
fffff800`3a10f790  00000000`00000000
fffff800`3a10f798  00000000`00000000
fffff800`3a076780  fffff800`38fa0cb8 nt!MiValidFault+0x204
fffff800`3a076788  fffff800`38fa71d9 nt!MiUserFault+0x325
fffff800`3a076790  fffff800`38fa793b nt!MmAccessFault+0x17b
fffff800`3a076798  fffff800`39226d4f nt!KiPageFault+0x3cf
fffff800`3a0767a0  fffff800`393c7e8f nt!NtAllocateVirtualMemory+0x5f
fffff800`3a0767a8  fffff800`3922b415 nt!KiSystemServiceCopyEnd+0x35
```

RTL Helpers

We defined our Copyin/Copyout functions, called RtlCopyToUser() and RtlCopyFromUser(). Each one:

- Ensure the integrity and range of the user and the kernel addresses
- Enable/Disable user access
- Read/Write
- Disable/Enable user access

In addition, we defined some Rtl helpers that wrap those calls to read/write specific types:

- RtlCopy{To,From}UserUlong()
- RtlCopy{To,From}UserUshort()
- RtlCopyToUserPtr()
- etc.

Moreover, since we have many cases of reading/writing different structures between user/kernel, we have generic functions for that:

```
#define RtlCopyStructToUser(Destination, Structure)
RtlCopyToUser(Destination, Structure, sizeof(*(Structure)), 1)
```

```
#define RtlCopyStructFromUser(Structure, Source, Alignment)
RtlCopyFromUser(Structure, Source, sizeof(*(Structure)), Alignment)
```

Challenges

Porting System Calls / Kernel-Mode Code

Many system calls and existing kernel code are pretty easy to convert into this new style. However, there are many places which are very complex and nontrivial to convert. Let's view a few examples.

NtQuery* system call family:

All of the NtQuery* system call family has a common pattern. Those system calls get as arguments:

- Userspace pointer – which points to some structure at userspace
- Class – which indicate the type of the structure

Then, they write all of the needed information to this pointer, and update another pointer (ReturnLength, a length variable which passed to the function by address) accordingly.

NtQueryInformationThread is no exception. It has a large switch-case, based upon the *ThreadInformationClass* argument. Each case scope in this switch treats the argument *ThreadInformation* as different structure, and we need to make all of those dereferencing from userspace safe.

One way to go here is to change all of the accesses to use the runtime accessors. But that's quite messy and will be hard to make all around the kernel. Another way to go here is to do it once in the end. Maintain during the entire system call a kernel mode buffer and write it all to user mode at the end. So, the end of the system call would look as follows:

…

```
    if (BufferToCopy) {
        if (PreviousMode != KernelMode) {
            try {
                RtlCopyToUser(ThreadInformation, BufferToCopy, BufferToCopySize,
ALIGNMENT_CHECKED);
            } except(EXCEPTION_EXECUTE_HANDLER) {
                Status = GetExceptionCode();
            }
        }
        else {
            RtlCopyMemory(ThreadInformation, BufferToCopy, BufferToCopySize);
        }
    }
```

Also, when we write return values to userspace, we need to use the appropriate accessors:

```
    if (NT_SUCCESS(Status) && ARGUMENT_PRESENT(ReturnLength)) {
        ULONG SizeToReturn = max(BufferToCopySize, BytesNeeded);

        if (PreviousMode != KernelMode) {
            try {
                RtlCopyToUserUlong(ReturnLength, SizeToReturn, 1);
```

```
            } except(EXCEPTION_EXECUTE_HANDLER) {
                Status = GetExceptionCode();
            }
        }
        else {
            *ReturnLength = SizeToReturn;
        }
    }

    return Status;
}
```

Many places in the kernel can be called from either a kernel-mode context (i.e. all buffers passed to the function are expected to be in the kernels virtual address space) or from user-mode (all buffers passed to the function are expected to be in the user-mode virtual address space). This can lead to a lot of boiler plate logic whereby buffers must be probed under circumstances and not probed under other circumstances. See the below code snippet for an example:

```
        case ThreadTebInformation:
            if (ThreadInformationLength != sizeof(THREAD_TEB_INFORMATION)) {
                return STATUS_INFO_LENGTH_MISMATCH;
            }

            if (PreviousMode != KernelMode) {
                try {
                    RtlCopyFromUser(&LocalTebInfo,
                        (PTHREAD_TEB_INFORMATION)ThreadInformation,
                        sizeof(LocalTebInfo),
                        ALIGNMENT_CHECKED);

                    if (PreviousMode != KernelMode) {
                        ProbeForWrite(LocalTebInfo.TebInformation,
                            LocalTebInfo.BytesToRead,
                            __alignof (UCHAR));
                    }
                } except(EXCEPTION_EXECUTE_HANDLER) {
                    return GetExceptionCode();
                }
                TebInfoPtr = &LocalTebInfo;
            }
            else {
                TebInfoPtr = (PTHREAD_TEB_INFORMATION)ThreadInformation;
            }
```

To avoid having to manually implement this logic in all places, we built accessor functions that accept the PreviousMode as a parameter. These accessor functions will probe prior to doing the memory access if necessary.

Page Deduplication

Some parts of memory manager, such as the page deduplication code, read from the user-mode virtual address space outside of an SEH exception handler and without probing.

In the case of the page deduplication code, the memory manager needs to scan the pages mapped in user-mode to see if multiple pages have the same content and can be merged in to a single physical page mapped at multiple virtual addresses.

This code doesn't probe because it is explicitly choosing user-mode virtual addresses. The code obtains locks to ensure that the user-mode virtual address being operated on cannot be paged out, swapped, or have it's virtual to physical mapping changed in some other way. Accordingly, no SEH exception handling is required.

Code like this does not fit in with the user-mode accessor model particularly well. Rather than update this code with user-mode accessors, we enabled and disabled user-mode access manually for this function.

Global User-Mode Structures

There are structures that are mapped in the user-mode virtual address space such as the TEB (Thread Environment Block) and PEB (Process Environment Block). These structures are quite large (for example, the TEB is several pages big) which makes them impractical to explicitly copy between user-mode and kernel-mode. At the moment, when the kernel updates these structures we manually enable/disable user-mode access.

Another more complicated example is PE image header parsing. The kernel may need to parse PE headers for PE files mapped in user-mode or kernel-mode. This process typically starts by providing the base address of the image, reading different fields from structures embedded in the image header, and doing pointer arithmetic based on the data read to calculate the address of the next header to parse.

The way the PE parser currently works is that if the base address passed in is in user-mode, all pointers computed while parsing the image must also be in user-mode. If the base address is kernel-mode, all pointers computed while parsing the image must also be in kernel-mode.

With SMAP enabled, all user-mode accesses must go through accessor functions. To keep the logic clean, we used the previously mentioned mode-based accessor functions which take a parameter which indicates if a probe needs to be done or not.

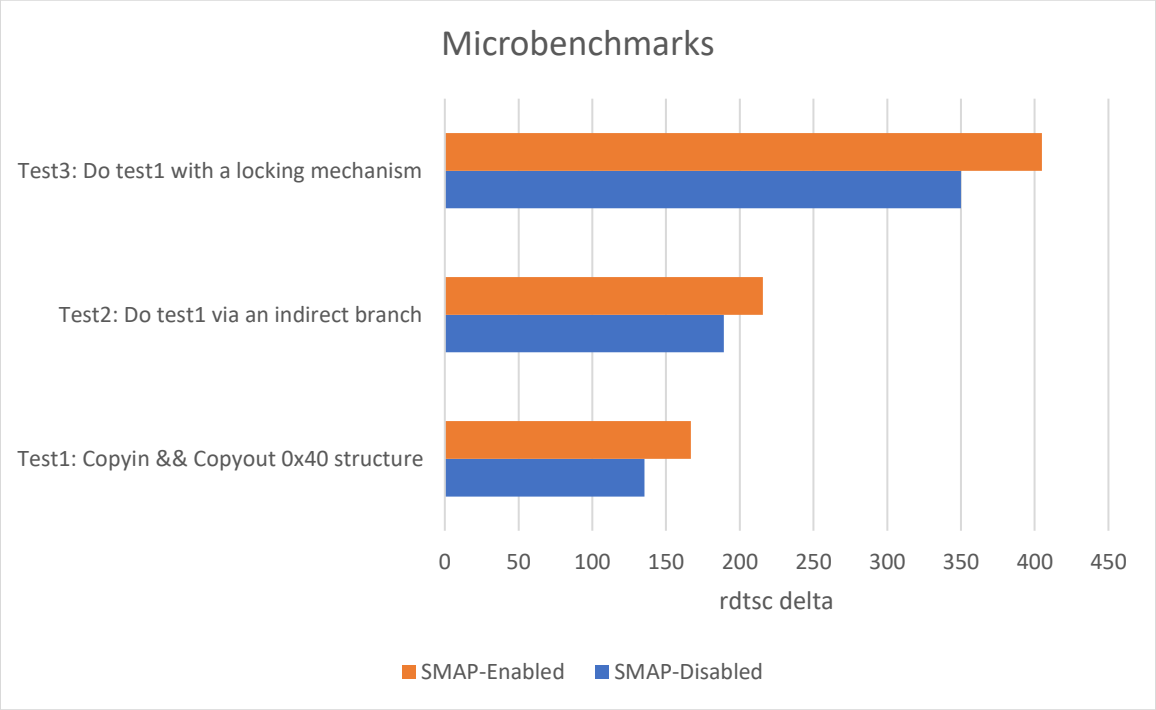
Performance tests

System Call Microbenchmarks

We created several microbenchmarks to test the performance of these new accessor functions with SMAP enabled. Of course, those microbenchmarks were tested with a non-instrumented build, to make sure we won't have noise in our tests. We enabled the instrumentation only on those specific functions. The microbenchmarks are built into a system call that is called from user-mode. We tested the following scenarios:

- Copyin / Copyout a 0x40 byte structure from/to userspace
- Call a function directly
- Call a function via a function pointer

The results of those microbenchmarks indicated a non-trivial regression on a bare-metal machine. Here are some numbers:



Those micro benchmarks have regression of ~23%. The fact that we are experiencing such performance hit shouldn't surprise us. Other operating systems that have SMAP enabled by default (such as Linux) were built in the copy_from_user/copy_to_user design in advance. So, the places where they have to do:

- Enable user access
- Do user access
- Disable user access

Are very specific and limited. The instrumentation mode being used by Windows requires that we have instructions to "enable user access" and "disable user access" in far more places than just the spots that are accessing user-mode.

Instrumentation check in the official build

While the microbenchmarks did not look promising, we wanted to get performance results for end-to-end scenarios under the assumption that all of NTOS was compiled with SMAP. If we enable SMAP for NTOS with the current compilation scheme, the only way we would be able to run end-to-end performance tests is:

1. Convert all of NTOS to use user-mode accessor functions – this would be too expensive to do for a simple performance test

2. Make the page fault handler re-enable user-mode access any time a SMAP violation occurs – this would have a significant impact on performance and would make the performance test results not actionable

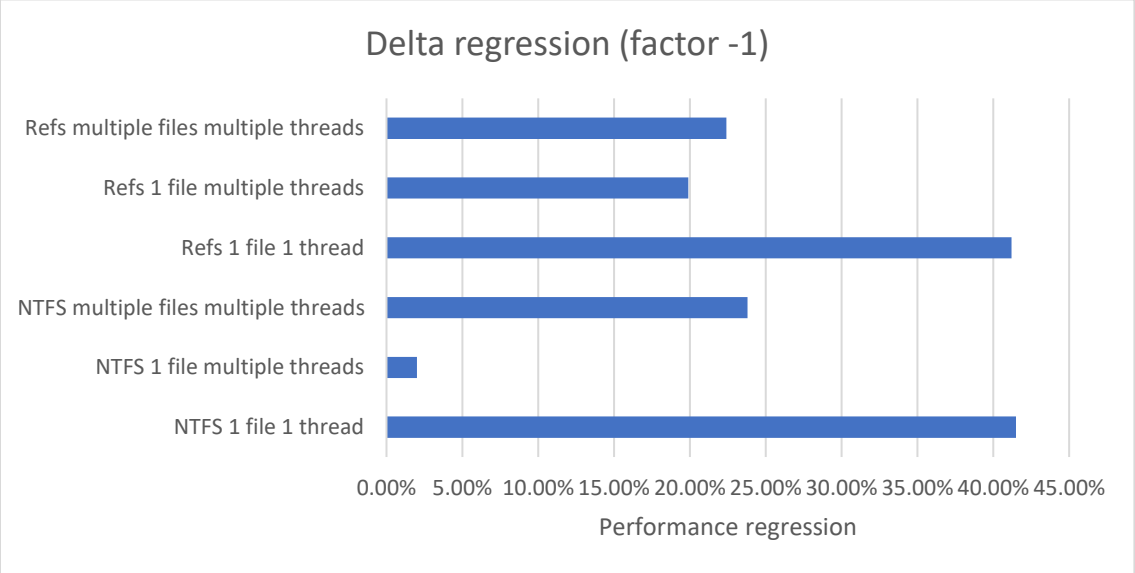
Instead, we introduced a new compiler flag which did a very simple but very nice trick:

The compiler does all the normal SMAP instrumentation. However, very late in compilation (after all optimizations have run) the compiler will swap out all instructions that "disable user-mode access" with instructions that "enable user-mode access". You end up with the same total number of SMAP instructions, all located at the same locations within the binary. However, we only use the enable user-mode access instructions so that the kernel never crashes.

This will let us check the performance hit that comes simply from having SMAP instructions compiled in to the binary. We make the assumption that there won't be much difference in that aspect between stac and clac instructions since:

- They are the same length (3 bytes each)
- They have same "consequences" with respect to caches, memory barriers, etc. based on our testing

Attached some performance regression in different operations in the filesystem code. This data represents the delta between operations per second against our parent branch:



It's important to note that other benchmarks had similarly large performance regressions. Unfortunately, these regressions are far too large to be viable to ship in Windows.