Arm Security Advisory Notice:

Armv8-M Secure Stack Sealing

Authors: Uma Maheswari Ramalingam,
Thomas Grocutt, Peter Smith

Introduction

This advisory offers a high-level overview of the software impact and mitigations for the vulnerability identified as CVE-2020-16273.

The scenarios that are discussed in this advisory relate to Armv8-M-based processors, including Cortex-M23, Cortex-M33, Cortex-M35P, and Cortex-M55, and any Armv8-M processors that are designed by partners under license that includes the Security extension that is called TrustZone. On these processors, if Secure software does not properly manage the Secure stacks when the stacks are created, or when performing non-standard transitioning between states or modes, for example, creating a fake exception return stack frame to deprivilege an interrupt, it is possible for Non-secure world software to manipulate the Secure Stacks, and potentially influence Secure control flow. This is not a hardware vulnerability. TrustZone solutions require both hardware and software. The scenarios that we discuss in this advisory are similar to other TrustZone-related software requirements.

Background

FNC_RETURN

When a Non-secure function is called from Secure state, it is necessary to hide and prevent modification of the return address to protect the Secure state information. Therefore, if a function call is marked as a branch from Secure to Non-secure, then the following actions are performed by the hardware:

- The return address and partial RETPSR values are pushed onto the Secure stack, as shown in Figure-1.
- LR is marked with FNC_RETURN value (`0xFEFF_FFFF or 0xFEFF_FFFE`).

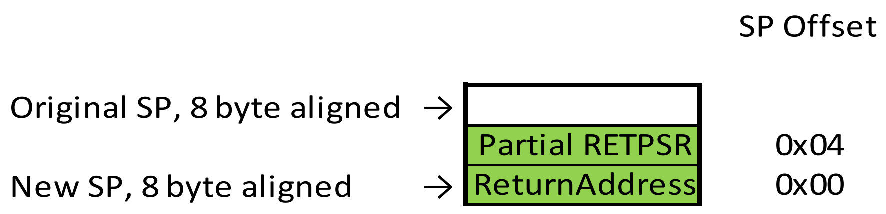

Figure-1

On a function return from Non-secure to Secure when a qualifying instruction, for example POP {pc}, loads PC with the FNC_RETURN value, the stacked contents, partial RETPSR and ReturnAddress, are read back from the Secure stack frame. During this function return process, a set of *integrity checks* are performed.

EXC_RETURN

Exception returns occur when a qualifying instruction, for example. POP {pc}, loads a value of 0xFFXXXXXX into the PC while in handler mode. The value that is written to the PC is intercepted and is referred to as the EXC_RETURN value to trigger the exception return process. On an exception return, several *integrity checks* are performed. These checks exist as a guard against errors in the system software and potential malicious behavior.

If the exception handler is Non-secure and the background code is Secure, the bottom of the Secure exception stack frame contains an integrity signature, as shown in Figure-2. The integrity signature, `0xFEFA125B or 0xFEFA125A`, is used to check the integrity of the stack frame during an exception return that switches the processor from Non-secure state to Secure state. If during the unstacking process it is found that the integrity signature does not match the architectural integrity signature value, then a `SecureFault` is generated. The stack frame type check flag, `SFTC`, at bit[0] of the integrity signature is checked against bit[4] of the EXC_RETURN value, and a `SecureFault` is generated if these two bits do not match. This fault generation prevents the Non-secure code from influencing how the stack frame is interpreted.

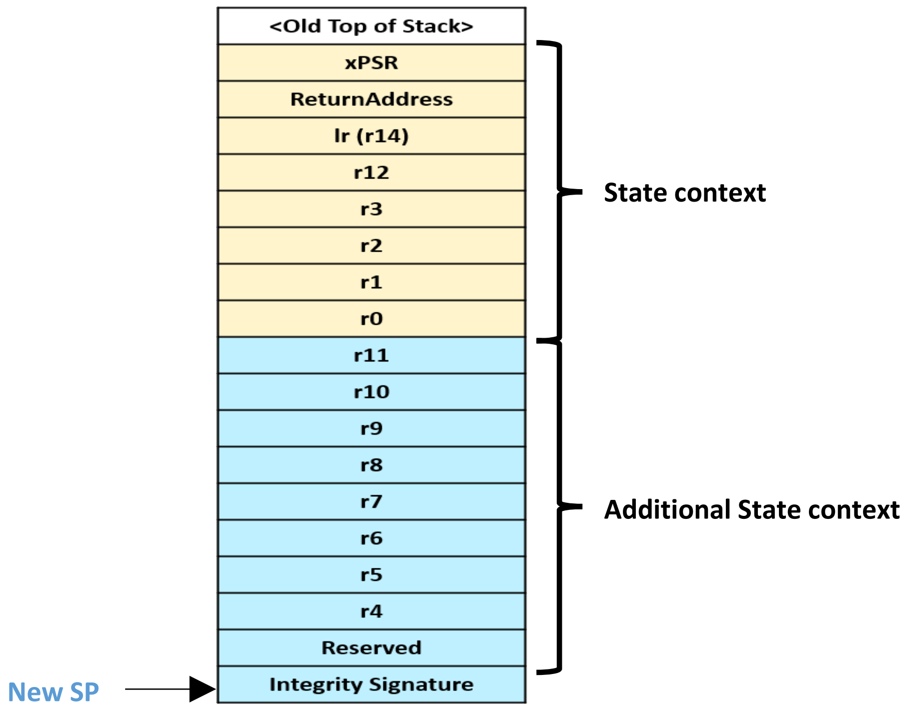

Figure-2

One important characteristic of the integrity signature is that it starts with 0xF but is not equal to FNC_RETURN or EXC_RETURN. This means that, in accordance with the Armv8-M architecture, any attempt to set the PC to this value will result in a `MemManage` fault. This is because addresses starting with 0xF are non-executable.

This behavior helps protect the system if an attacker triggers a return to Secure world using an FNC_RETURN instead of an EXC_RETURN. In that scenario, the integrity signature will be interpreted as the return address of an FNC_RETURN stack frame, triggering a fault . Also, because the integrity signature is a non-executable address, it is not possible for it to ever appear in the return address of an FNC_RETURN stack frame. Therefore, if an attacker does an EXC_RETURN instead of an FNC_RETURN, or if an attacker does an FNC_RETURN instead of an EXC_RETURN, Armv8-M-based processors will always fail with a Secure HardFault or a Secure MemManage fault.

Secure Stack Pointer software vulnerability issue

In the Armv8-M architecture, there are two stack pointers in Secure state: the Process Stack Pointer, PSP_S, and the Main Stack Pointer, MSP_S. In normal circumstances, when a Processing Element (PE) switches from the Secure world to the Non-secure world due to a function call or an exception, the currently-selected stack should contain either a function return stack or an exception stack frame. A return from Non-secure world to Secure world using the same Secure stack pointer is protected. However, it is possible for the Non-secure world to manipulate the execution state of the PE. This manipulation could cause the return process to use a different stack pointer.

If a Secure stack is empty, for example when there are no active exceptions, the Main stack may be empty, so an attacker could potentially use a fake EXC_RETURN or FNC_RETURN operation to trigger a stack underflow scenario. Because the memory contents just above the stack memory could be any value, including zeros or random value, there is a possibility for the data word above the stack to match a stack frame integrity signature or a Secure executable address value, potentially causing incorrect code execution through a stack underflow scenario.

Here is one example scenario of a stack underflow situation:

Note: In the real world, there could be other scenarios where a stack underflow situation is caused. This means that software developers need to carefully analyze stack pointer usage in their system.

Scenario 1

As shown in Figure-3, consider a PE executing in Secure Thread mode with the Secure Process Stack Pointer (PSP_S) as the current stack pointer when a call is made to a Non-secure function. In the Non-secure function, an SVC instruction is executed causing a transition to a Non-secure SVC handler. If the handler branches to a fake FNC_RETURN value with a BX <reg> instruction, then it is possible that the PE shall proceed to pop back ReturnAddress and RETPSR values from a potentially empty Secure Main Stack Pointer (MSP_S).

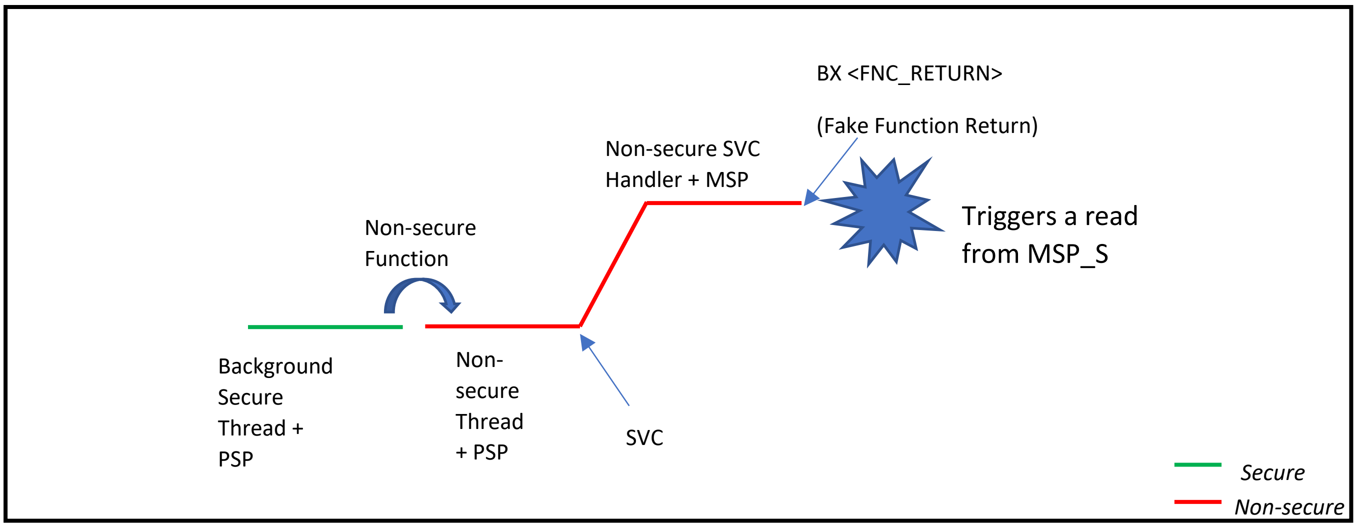

Figure-3

This means that a malicious or incorrect Non-secure world code can cause the PE to pop back from the wrong stack and use incorrect values for return address and RETPSR. If ReturnAddress can be influenced by the Non-secure world, then this provides an attack vector to jump to an address in Secure world.

Mitigation for Scenario 1

To ensure that stack underflow attacks like the one mentioned in Scenario 1 are detected and stopped, the Secure software developer should reserve two words of stack memory, and place a special value 0xFEF5EDA5 just above the real stack space, as shown in Figure-4. Two words of stack space are needed to keep the stack double word aligned. The special value used is architecturally guaranteed to not match the stack frame integrity signature or a function address, because the address range 0xE0000000 to 0xFFFFFFFF is non-executable. This technique is called Stack Sealing.

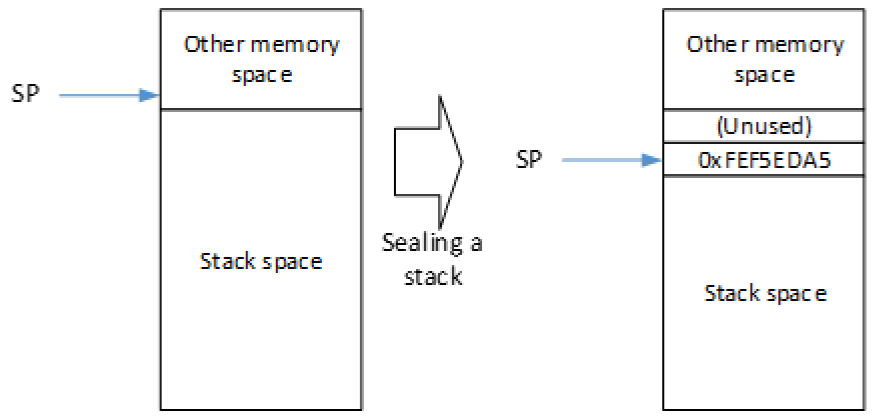

Figure-4

To ensure compatibility with future versions of the Arm architecture, Arm recommends that the value used to seal the top of the stack is 0xFEF5EDA5.This value has the following properties:

- It is not a valid FNC_RETURN or EXC_RETURN value.
- It is not the integrity signature, so would not inadvertently mark the stack as containing a valid exception stack frame.
- The value starts with 0xF and is therefore not a valid instruction address.

Arm strongly recommends sealing both the Process stack and Main stack of the Secure software. This recommended sealing value, 0XFEF5EDA5, has been deliberately picked for these properties Therefore, sealing the top of the stack prevents attacks that can be triggered by faking an EXC_RETURN or FNC_RETURN.

Scenario 2

As a part of a TrustZone solution, a generic system might be divided into a Secure Processing Environment (SPE) for the sensitive assets and the code that manages them, and a Non-secure Processing Environment (NSPE), in which the main application and communication firmware executes, as you can see in Figure-5.

Within the SPE, a Trusted component may be responsible for subdividing the SPE into multiple isolated partitions and providing communication between the SPE and the NSPE. The Secure Partition Manager (SPM) manages the independent Secure partitions and provides hardware-enforced compartments for individual code blocks, by limiting access to memories and peripherals.

One of the major responsibilities of the SPM is to forward and deprivilege interrupts to the unprivileged handler, operating in Secure Thread mode, that has been registered for them. For example, if there is an interrupt targeting a Secure partition, Partition-1 in Figure-5, then the SPM registers this interrupt request before giving the control to the Secure partition to execute in unprivileged mode.

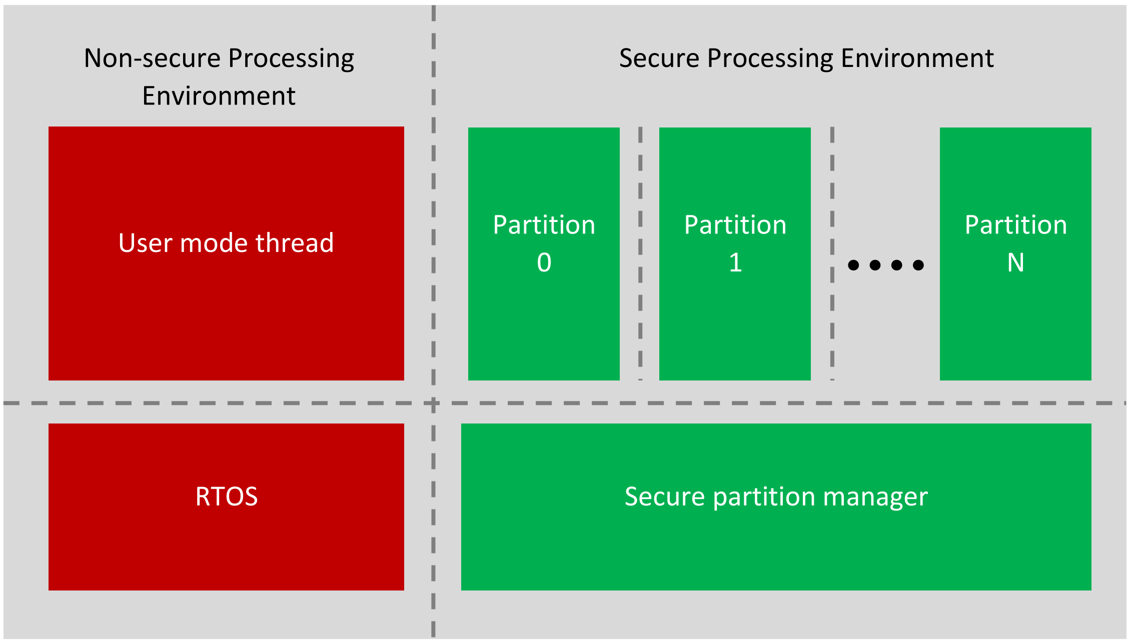

Figure-5

To enable Secure services of peripherals where some of the peripherals communicate using interrupts, it is important to provide Secure partitions as a way to handle interrupts along with isolation levels. To keep the Secure software isolation levels intact, partitions are isolated using the concept of deprivileged interrupts.

Consider a scenario as shown in Figure-6 that involves following points:

- In a privileged Secure thread mode, background code sequence, configure the priority of IRQ to have lower priority than SVC.
- Trigger and enter Secure IRQ handler, which acts as a wrapper to create an unprivileged code container. Save Callee registers and clear contents of registers in IRQ handler. It may also be necessary to context switch any other state that is required to run the unprivileged context.
- Execute an SVC instruction to deprivilege the execution. Enter SVC handler and do all the context saving that is required. The execution context at the time of SVC execution will be pushed on to MSP_S. Modify CONTROL.npriv to be 1. In SVC handler, construct a fake exception return stack frame, EXC_RETURN to enter an unprivileged Secure thread mode using Process Stack Pointer (PSP_S).

- On SVC exception return, we are in Unprivileged Secure thread mode with PSP as current stack pointer, but still running with the execution priority of the IRQ, so only a higher priority interrupt can pre-empt. This case can also be considered as a one of the sandboxes executing in Secure unprivileged thread mode. At this stage, if a Non-secure interrupt causes pre-emption, or a Non-secure Function Call back is invoked, then it is possible that that Non-secure world could return back to Secure world using FNC_RETURN with the use of MSP_S as the current stack pointer.

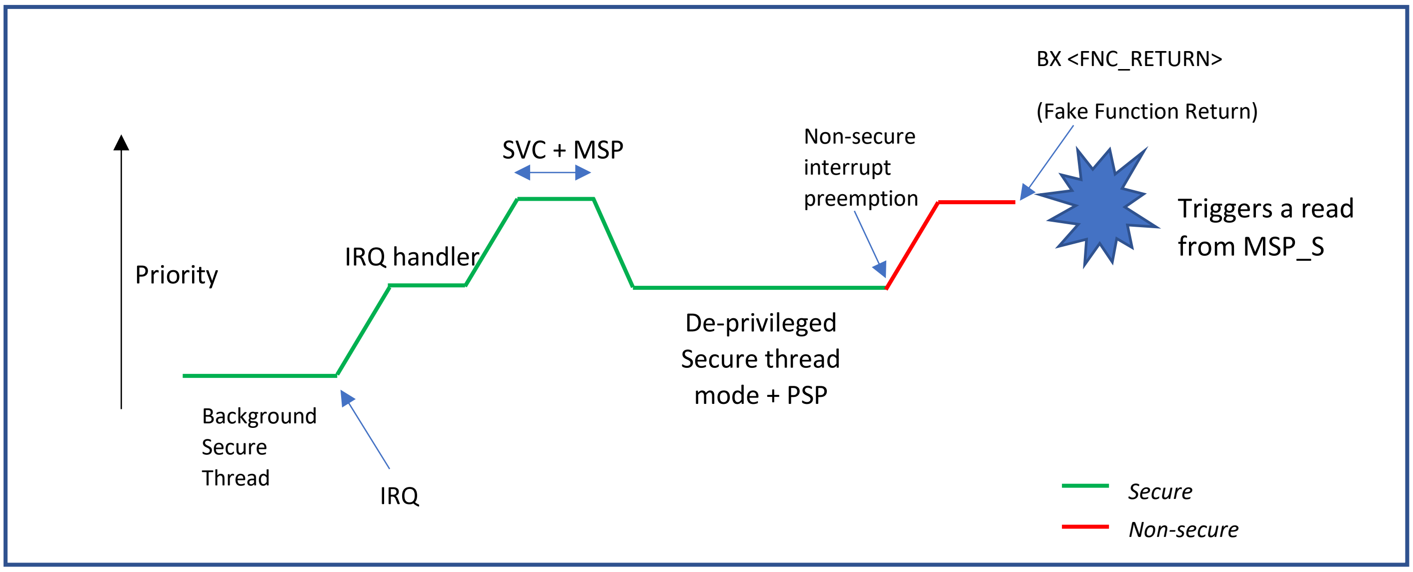

Figure-6

This is another scenario where malicious or incorrect Non-secure code can cause the PE to pop back from the wrong stack, and Non-secure world can influence a branch to Secure address.

Mitigation for Scenario 2

In this process of deprivileging an interrupt, be careful when creating a fake stack frame in the SVC handler. Be sure that both the Secure Main stack pointer and the Secure Process stack pointers are sealed before entering deprivileged Secure thread mode, to ensure that we obtain a guaranteed fault if there is an attempt to trigger an incorrect type of return operation. This is because the stack frame is manually created by the software developer and is not created automatically by the hardware.

Conclusion

Although hardware defends against many incorrect transitions between Non-secure to Secure world, through integrity checks, hardware cannot automatically defend against cases where software manually manipulates the stacks or state transitions. In this advisory, we have shown a few examples in which software could change the Secure stack pointer manually to potentially cause a stack underflow security vulnerability. In Scenario 1, we have shown that stack underflow security vulnerabilities can be mitigated using the Stack Sealing technique. In Scenario 2, we have also shown a simple scenario where the stack frame is manually constructed, in which sealing the stack pointers plays a vital role in preventing software vulnerability.

References

1. **Armv8-M Architecture Reference Manual**

Appendix

Configuring stack limit registers

If the Armv8-M Main Extension is implemented in a system, then to protect against overflow attacks on a Secure stack pointers, Arm recommends configuring two stack limit registers, MSPLIM_S and PSPLIM_S, that limit the extent to which the Main and Process secure stack pointers can descend. Architecturally, a program violating this stack limit will result in a synchronous fault. See Section: B3.21 of the **Armv8-M Architecture Reference Manual** for more details.

The following code shows a simple stack limit configuration for both Main and Process Stack pointer.

```
==============================================================
    // Get Stack Limit Value
    ldr r0,=<address of stack base>
    // Set Main and Process stack limit registers.
    msr MSPLIM, r0
    msr PSPLIM, r0
==========================================================
```

Note: On Arm Compiler 6, if you have used the scatter file method of placing the stack using an execution region called `ARMLIB_STACK` or `ARMLIB_STACKHEAP`,then the linker-defined symbols `Image$$ARMLIB_STACK$$Base` or `Image$$ARMLIB_STACKHEAP$$Base` linker defined can be used to find the stack limit.

Programming Secure MPU regions

If an optional Secure MPU is implemented in a system, Secure firmware can utilize the MPU to define memory access permissions and memory attributes for different regions within the 4GB memory by programming its MPU regions. Depending upon the type of access, for example Instruction fetch or Data access, the MPU helps in monitoring the accesses, and can trigger a synchronous fault exception if unauthorized access is attempted. MPU can define regions of memory where access is never allowed by instruction fetches, thus preventing any potential malicious code from being executed from those regions. MPU can prevent stack overflows in one task from corrupting memory belonging to another task.

Software developers can use CMSIS-Core API functions to program MPU regions. These functions are available at **https://arm-software.github.io/CMSIS_5/Core/html/group__mpu8__functions.html**

More information about the MPU in the Armv8-M architecture is available from this application note: **Memory Protection Unit (MPU) Version 1.0**

Stack sealing with Arm Compiler 6

As discussed in this advisory, Arm recommends that both the Secure Process and the Secure Main stack pointer have a special value, 0xFEF5EDA5, added to the top of the stack. This prevents an attacker using a fake FNC_RETURN or EXC_RETURN to enter the Secure world.

Arm Compiler 6 comes with a choice of two C libraries:

- Microlib
- Standardlib

When using these libraries, the initialization code will set the Main stack pointer before entering main. This will not seal the Main stack pointer and will not set the Process stack pointer. To fulfill the Stack Sealing recommendations, a program must seal the Main stack and the Process stack to mitigate this vulnerability. If the Process stack is not used by Secure state, Arm recommends that it is set to the initial value of the Main stack pointer, so that sealing the Main stack also seals the Process stack. The sealing must occur before any potential transition to Non-secure state.

Sealing the Main Stack Pointer

Although there are other possible ways to seal the stack pointer, the two main variations recommended by Arm are:

- Place the seal value after the stack in memory through scatterloading. The seal value will be copied into place by the scatterloading code before the stack pointer is initialized.
- Call a function that executes before main is entered, to configure a seal value onto the top of the stack.

Setting the Process Stack Pointer

To set the Process stack pointer requires an MSR instruction with PSP as the destination executed from Privileged mode. This must be performed before entering main. This can be combined with the code sequence that will seal both the Main stack pointer and Process stack pointer.

Example of scatter file to seal the stack

```
=========================================================
LR1 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
    # We reserve 8 bytes of stack for the seal
    ARM_LIB_STACKHEAP 0x10000000 EMPTY (0x10000 - 8)
    {

    }
    # Seal is placed immediately after the stack.
    SEAL +0
    {
        # The +FIRST prevents .seal from being removed as unused.
        *.o(.seal+FIRST)
    }
}
=========================================================
```

The seal can be defined in assembly language as:

```
=========================================================
// Data section that the linker will place at the top of the Main stack
 .section .seal, "a", %progbits
 .word 0xfef5eda5
 .word 0xfef5eda5
=========================================================
```

Using a function to seal stack pointers

The method to insert a user-defined function call depends on the library that is being used.

Standardlib

When defined, the function `_platform_post_stackheap_init` is called by `__rt_entry` after the stack pointer has been initialized. By defining this function, we can seal the stack and set up the Process stack pointer before it is used by any other library function.

If the scatter loading method is used to seal the stack, only the code to set the Process stack pointer is required. This code starts at the comment:

```
===============================================================
// Initialise the PSP to the same value of SP. This ensures that the PSP
// stack is also sealed. If the program later changes the location of PSP
// it must seal the PSP stack at that point.
===============================================================
```

Example code for `_platform_post_stackheap_init`

```
===============================================================
    .text
    // __platform_post_stackheap_init is called after the C-library
    // initialization code has set the initial value of sp. We assume:
    // - sp has {r0, r1} pushed from __rt_entry caller
    // - the Process stack pointer has not been initialized.
    // - this function executes in handler mode.
    // At the end of the function the stack frame will look like:
    // Initial stack pointer             |0xfef5eda5| (contents of r1)
    //                                   |0xfef5eda5| (value to seal stack)
    //                                   |r1        | (from caller)
    //                                   |r0        | (from caller)
    // Main and Process stack pointers -> |??????????|

    .global _platform_post_stackheap_init
    .type _platform_post_stackheap_init, %function
_platform_post_stackheap_init:
.L1:
    // __rt_entry_postsh_1 pushes r0 and r1 onto the stack before calling
    // _platform_post_stackheap_init. We must seal above r0,r1 so that the
    // __rt_entry_postsh_1 pops the r0 and r1 it is expecting.
    pop         {r0,r1}
    movw        r2, #0xeda5
    movt        r2, #0xfef5
    mov         r3, r2
    // r2,r3 = 0xfef5eda5
    // seal stack while maintaining 8-byte stack alignment, pushing r0, r1
    // back on so that __rt_entry_postsh_1 can pop them.
    push        {r0,r1,r2,r3}
    // Initialise the PSP below the sealed stack. This ensures that the PSP
    // stack is also sealed. If the program later changes the location of PSP
    // it must seal the PSP stack at that point.
    add r0, sp, #8
    msr PSP, r0
    bx lr
    .size __platform_post_stackheap_init, . - .L1
==============================================================
```

Microlib

There is no direct equivalent of _platform_post_stackheap_init in microlib. Instead we use $Sub$$ and $Super$$ to intercept the call to a function that runs before entering main. The function to intercept depends on whether the stack is in zero-initialized memory, or whether run-time static initialization is performed.

If the stack is not in zero-initialized memory, then the best function to intercept is the __scatterload function. This is because it is called after the stack pointer is initialized so that we can seal the stack before any other code that uses the stack is called.

If the stack is in zero-initialized memory, then a function that runs after __scatterload must be intercepted because __scatterload is responsible for zeroing the memory. There are two candidate functions to intercept __cpp_initialize__aeabi_ and the main function itself. The function __cpp_initialize__aeabi_ is responsible for running the programs C++ static constructors and will only be present in the program if it is needed. Arm recommends intercepting __cpp_initialize__aeabi_ if it is present, so that the stack is sealed before constructors are run. If __cpp_initialize__aeabi_ is not present, then the only choice is main.

The following example intercepts __scatterload. To intercept one of the other functions, just replace __scatterload with the function name. For example, if we are intercepting main then $Sub$$__scatterload becomes $Sub$$main.

```
==============================================================
    // $Sub$$__scatterload intercepts the branch to __scatterload
    // The library initialization code has completed:
    // - sp is set to the top of the Main stack pointer.
    // - the Process stack pointer has not been initialized.
    // - this function executes in handler mode.
    // At the end of the function the stack frame will look like:
    // Initial stack pointer              |??????????| (contents of r1)
    //                                    |0xfef5eda5| (value to seal stack)
    // Main and Process stack pointers -> |??????????|

    .global $Super$$__scatterload
    .global $Sub$$__scatterload
    .type $Sub$$__scatterload, %function
$Sub$$__scatterload:
.L1:
    movw        r0, #0xeda5
    movt        r0, #0xfef5
    mov         r1, r0
    // r0, r1 = 0xfef5eda5
    // seal stack while maintaining 8-byte stack alignment
    push        {r0,r1}
    // Initialise the PSP to the same value of SP. This ensures that the PSP
    // stack is also sealed. If the program later changes the location of PSP
    // it must seal the PSP stack at that point.
    mov r0, sp
    msr PSP, r0
    // Call __scatterload
    b.w $Super$$__scatterload
    .size $Sub$$__scatterload, . - .L1
==============================================================
```

Note: If armasm assembler is used in your projects, then for migration guidelines refer to **ARM Compiler Migration and Compatibility Guide**.

Stack sealing with GNU Toolchain

When using the GNU toolchain for embedded processors, Arm recommends using a function that seals the Main stack and sets up the Process stack. The default crt0 code in newlib does not set the Process Stack Pointer.

In this section, two examples are provided as a reference that can be used to set both Main and Process stack pointer. If you would like a separate Process stack, then you should make sure that it is also sealed in the same way.

Example 1: Overriding _stack_init

The following example is only available for version 9-2020-q2-update and newer. This toolchain is packaged with a newlib version where the `_mainCRTStartup` function calls to a '.weak' defined `_stack_init`, right after SP is set. We can provide a non-weak definition of `_stack_init` that overrides the version in the library. In our definition we can seal the Main stack and set PSP.

Beware that `_stack`_init is called before 'memset', meaning that if your stack resides in the .bss region, memset will overwrite the stack seal with zeroes, which renders the mitigation useless.

The beginning of this example code was copied from the existing newlib `_stack_init` definition. Make sure to define ARM_RDI_MONITOR if you are using semihosting.

```
==============================================================
Example code for _stack_init
#if __ARM_ARCH_ISA_THUMB == 1 && !__ARM_ARCH_ISA_ARM
#define THUMB1_ONLY
#endif
.text
.syntax unified
.thumb
.global _stack_init
.thumb_func
_stack_init:
 /* Set SL register.  */
#if defined (ARM_RDI_MONITOR) /* semihosting */
        cmp     r2, #0
        beq     .Lsl_forced_zero
        /* Allow slop for stack overflow handling and small frames.  */
# ifdef THUMB1_ONLY
```

```
        adds    r2, #128
        adds    r2, #128
        mov     sl, r2
# else
        add     sl, r2, #256
# endif
.Lsl_forced_zero:

#else /* standalone */
        /* r3 contains SP for System/User mode. Set SL = SP - 0x10000.  */
#ifdef THUMB1_ONLY
        movs    r2, #64
        lsls    r2, r2, #10
        subs    r2, r3, r2
        mov     sl, r2
#else
        /* Still assumes 256bytes below SL.  */
        sub     sl, r3, #64 << 10
#endif
#endif
      /* Code to seal the stack.  */
      movw  r1, #0xeda5
      movt  r1, #0xfef5
      mov   r2, r1
      push  {r1, r2}
      mov   r1, sp
      msr   PSP, r1
      bx    lr
==============================================================
```

Example 2

You can call a function in the main function of your secure state program to seal PSP and MSP before your first call to Non-secure state. However, you have to make sure that your Secure state boot code does not have any transition to Non-secure state, and that Non-secure state interrupts are not enabled before sealing the stack. The following code shows an example of this type of function:

```
==============================================================
#include <stdint.h>
extern void * __stack_base__;
void
seal_stack (void)
{
  unsigned seal_value = 0xfef5eda5;
  asm ("mov r0, %[seal_value] \n\t"
          "mov r1, r0  \n\t"
          "stmdb %[sb_p_8]!, {r0, r1} \n\t"
          "msr PSP, %[sb_p_8]"
          : : [seal_value] "r" (seal_value), [sb_p_8] "r"(((intptr_t)__stack_base__) +
8) : "memory", "r0", "r1");
}
==============================================================
```