Arm Security Advisory
ASA-010

Version: 1.0

Security weakness in PCS for CMSE

Confidential

Security weakness in PCS for CMSE

Release information

Document history

Version	Date	Confidentiality	Change
1.0	13/02/2024	CONFIDENTIAL	First Published

Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

Web Address

http://www.arm.com

Contact

psirt@arm.com

Contents

1 Introduction

This security weakness relates to procedure calls between non-secure and secure states when using the Cortex®-M Security Extensions (CMSE). You might be affected if:

- Toolchain user: you develop code for Armv8-M secure state and use CMSE-compliant procedure calls to or from non-secure state and you pass argument or return types of size less than 32-bits. See Information for Toolchain Users for further details.

- Toolchain developer: your toolchain implements support for CMSE-compliant procedure calls. See Information for Toolchain Developers for further details.

- Information for Toolchain Users for further details.
- Toolchain developer: your toolchain implements support for CMSE-compliant procedure calls. See Information for Toolchain Developers for further details.

2 Issue description

The Armv8-M architecture for microcontrollers defines an optional Security Extension. The Security Extension is designed to combine code from multiple vendors without requiring trust between them. It achieves this by partitioning processor state and memory into Secure and Non-secure states and provides controlled mechanisms to transfer execution and data between these.

To make the features of the Security Extension accessible to software developers, the *Cortex-M Security Extensions* (CMSE) defines C language support to place code and data in Secure and Non-secure states and to make function calls between states.

Software written to the guidelines in ARMv8-M Secure software guidelines 2.0 using tools that implement Arm v8-M Security Extensions Requirements on Development Tools separate Secure state and Non-secure state:

- Non-secure state can only call functions in Secure state that have veneers in the Non-secure Callable region that forward control flow to an entry function in Secure state. Non-secure state can pass data to Secure state via function parameters. The Non-secure state code for a function call follows all the standard AAPCS32 rules.

- Secure state may call functions in Non-secure state via a BLXNS instruction. The Non-secure state function may return a value to Secure state. The Non-secure state functions called from Secure state follow all the standard AAPCS32 rules.

In normal operation, Non-secure state follows all the AAPCS32 rules when calling entry functions. All integral types with a size less than a word are zero or sign extended to a word. Return values from Non-secure functions called by Secure state are also zero or sign extended when required by the AAPCS32.

If Non-secure state is compromised by an attacker, then Secure state functions may be called with arguments, or Non-secure functions may return values, that are not zero or sign extended. To perform an attack via calling an entry function an attacker must have the following capabilities:

- Ability to set the arguments of function calls. For example, via a gadget that sets one of the 4 argument registers r0, r1, r2 or r3 to a value of the attacker's choice.

- Ability to call the Non-secure gateway veneer for the entry function. For example, via a ROP or JOP gadget using the address of the Non-secure gateway veneer or targeting a direct function call to the Non-secure gateway after the sign or zero extension of parameters.

To perform an attack via a return value requires the attacker to substitute a function that Secure state is calling with a malicious implementation. This may occur if an attacker does not have access to Secure state but has compromised the integrity of Non-secure state.

3 Impact

An attacker who can pass out-of-range values to code executing in Secure state might be able to cause incorrect operation in Secure state, for example:

- An out-of-range value used as an array index might allow unbounded memory accesses to occur (CWE-119).
- An out-of-range value used in a calculation might allow incorrect results to be produced (CWE-682).

The exact impact cannot be determined without examination of the secure code and how it processes the affected type. For this reason, Arm is not publishing a CVSS score for this issue.

4 Information for Toolchain Users

This information is for toolchain users who are developing secure code using CMSE. It is assumed the reader is familiar with the Armv8-M security model and how C source code maps onto this using the procedure call standard.

4.1. Is my program affected?

4.1.1 Required conditions

The following conditions must all be met for the program to be at risk of being affected:

- The program runs in Secure state on an Arm CPU that implements the Security Extension, also known as Arm TrustZone for Armv8-M.
- The program follows the CMSE standard, using Non-secure entry functions as entry points to Secure state, and Non-secure calls for calls to Non-secure state from Secure state.
- Integral types of less than word size (32-bits) are passed as arguments to entry functions or are returned from Non-secure functions called via Non-secure calls from Secure state. See Affected types for further details.
- There is a path through Secure-state where having an out-of-range value in one of the affected arguments or return values can cause a Denial of service or incorrect operation of Secure state.

4.1.2 Impact of an out-of-range value

In many cases an out-of-range parameter or return value will not lead to incorrect operation of Secure state. For example, an existing bounds check may catch out-of-range values. Due to the variability of compiler optimizations, such as those that remove bounds checks based on the range of values a type can represent, Arm recommends that the disassembly of the secure code is studied to trace the impact of out-of-range values.

The following is a non-exhaustive list of problems that could occur.

4.1.3 Out of bounds accesses

The compiler may use information about the type to optimize away bounds checks.

```
#include <arm_cmse.h>
#define ARRAY_SIZE (256)

char array[ARRAY_SIZE];

char __attribute__((cmse_nonsecure_entry))
secureFunction(unsigned char index) {
```

```
  // Compiler may optimize away bounds check as value is an unsigned char.
  // According to AAPCS32 caller will zero extend to ensure value is < 256.
  if (index >= ARRAY_SIZE)
    return 0;
  return array[index];
}
```

Bounds checks that cannot be inferred from the type are not optimized away. For example:

```
char __attribute__((cmse_nonsecure_entry))
secureFunction(unsigned char index) {
  // Out of range values are present within range of unsigned char, bounds
  // check cannot be removed based on type alone.
  if (index < 1 || index > 5) {
    // invalid value, report error message
  }
  ...
}
```

4.1.4 Overflow checks

The Cert C coding standard requires that integer expressions are guarded against overflow. A modification of the Compliant Solution adapted for short types is:

```
#include <limits.h>

void f(signed short ss_a, signed short ss_b) {
  signed short sum;
  if (((ss_b > 0) && (ss_b > (SHRT_MAX - ss_b))) ||
      ((ss_b < 0) && (ss_a < (SHRT_MAX - ss_b)))) {
    // Overflow detected
  } else {
    sum = ss_a + ss_b;
  }
  ...
}
```

This overflow check depends on ss_a and ss_b being signed short values. Out of bounds values can overflow the `SHRT_MAX - ss_b` and not get caught by the overflow check.

4.1.5 Switch statement

A switch statement with a case for each of the values in a type and no default value can be implemented by a jump table. As every value permitted by the type has a case the range check can be optimized away. For example:

```
unsigned char f(unsigned char x) {
  // All possible values of x according to the fundamental type of Unsigned
  // byte have a case statement.
  switch(x) {
    case 0:
      return 0;
    case 1:
      return 1;
    ..
    case 25
      return 255;
  }
}
```

Such a switch statement may be implemented as a jump table, for example:

```
movw    r1, :lower16:.Lswitch.table.f
movt    r1, :upper16:.Lswitch.table.f
// r0 is parameter x r1 = base of table
ldrb    r0, [r1, r0]
```

With an out-of range `x`, the table may read outside the bounds of the branch table, potentially leaking information from Secure state or crashing the program leading to a denial of service.

4.2. Affected toolchains

The table below shows the known toolchains that can generate code with the weakness in the Affected Versions column. The Fixed Versions column includes updated tools that do not generate code that is affected by the weakness.

Toolchain	Affected Versions	Fixed Versions
Arm Compiler for Embedded	6.13 – 6.21	6.22 (planned)
Arm Compiler for Embedded FuSa	6.16 all versions.	None available
clang	Clang 11 – Clang 18 Also includes any compiler that supports CMSE that is based on llvm technology from LLVM 11 – LLVM 18	Clang 19 (planned)
GCC	GCC 10 – GCC 13	GCC 14 (planned)

Developers who are using other toolchains should contact their toolchain vendor to determine whether they are impacted and about the availability fixes.

4.3. Affected types

This section describes the types that may be affected by this weakness.

The AAPCS32 in sections Data Types and Alignment has a table of Fundamental Data Types giving the byte size and alignment of each of the types. For this weakness, all types have a Type Class of Integral. The mapping of C and C++ built-in data types to the Fundamental Data Types is given in another table Arithmetic Types. The table below shows the integral Fundamental Data Types, their mapping to C and C++ built-in data types, and whether they are affected by the weakness.

Fundamental Type	Equivalent C/C++ Built-in Type	Size in bytes	Affected
`Unsigned byte`	`char, unsigned char, bool, __Bool`	1	Yes
`Signed byte`	`signed char`	1	Yes
`Unsigned half-word`	`unsigned short`	2	Yes
`Signed half-word`	`short`	2	Yes
`Unsigned word`	`unsigned int, unsigned long`	4	No
`Signed word`	`int, long`	4	No
`Unsigned double-word`	`unsigned long long`	8	No
`Signed double-word`	`long long`	8	No

4.3.1 Enumerated types

Enumerated types like a C/C++ `enum,` when implemented to strictly conform to the AAPCS32 use a signed word fundamental type. A common procedure call variant implemented by armclang, clang and GCC is `-fshort-enums` which uses the smallest possible integral data type that can represent the values of the enum. For example, an enum with values between -128 and +127 can be represented by the Signed Byte integral type.

- Programs that use `-fshort-enums` must treat the enumerated type as the smallest integral type that can represent the values in the enumeration.
- Programs that use `-fno-short-enums` do not need to consider enumerated types as the smallest integral type used in this case is a Signed word.

4.3.2 Wide characters wchar_t

The AAPCS32 preferred integral type for `wchar_t` is Unsigned word. Like enumerated types, armclang, clang and GCC have an option called `-fshort-wchar` that uses Unsigned half-word instead.

Programs that use `-fno-short-wchar` do not need to consider wide characters as the smallest integral type used in this case is an Unsigned word.

4.3.3 _BitInt(N)

`_BitInt(N)` is a C2X extension to provide a Fundamental Type for N-bit integers. Uses of `_BitInt(N)` or `usigned _BitInt(N)` where `N <=64` are mapped to the smallest integral type where byte-size of the integral type * 8 >= N. Larger values of N are assigned to arrays of fundamental types. The table below shows the mappings of _BitInt(N) where N <= 64 to the integral types and whether they are affected by the weakness.

N-bit integer type	Fundamental Type	Size in Bytes	Affected
_BitInt(N) : N <= 8	Signed byte	1	Yes
unsigned _BitInt(N) : N <= 8	Unsigned byte	1	Yes
_BitInt(N) : 8 < N <= 16	Signed half-word	2	Yes
unsigned _BitInt(N) : 8 < N <= 16	Unsigned half-word	2	Yes
_BitInt(N) : 16 < N <= 32	Signed word	4	No
unsigned _BitInt(N) : 16 < N <= 32	Unsigned word	4	No
_BitInt(N) : 32 < N <= 64	Signed double-word	8	No
unsigned _BitInt(N) : 32 < N <= 64	Unsigned double-word	8	No

4.3.4 Other type classes such as Floating Point and Aggregates

Only integral types are affected. Other type classes are not affected, including those smaller than a word. This includes half-precision floating point values which are smaller than a word. It also includes aggregate types such as C++ structs and classes that contain integral types that are smaller than a word.

4.4. Software mitigations

4.4.1 Recompile secure state with updated tools

Use updated versions of Arm Compiler for Embedded (armclang), clang, and GCC that generate code conformant to the updated ACLE CMSE specification. Code-generation for entry functions has the following changes:

- Parameters of entry functions that are of integral type and size less than a word are narrowed so that values are within the range of the integral type.
- Return values of non-secure state functions called from secure state that are of integral type and size less than a word are narrowed so that values are withing the range of the integral type.

These changes do not change the API or ABI, and only need to be applied to Secure state. No changes are required to Non-Secure state.

4.4.2 Change API between Secure and Non-Secure state

If updated tools are unavailable or cannot be used, the weakness can be avoided by changing the API.

The weakness only applies to function parameters and return values of an integral type with size less than a word. If the API between Secure and Non-secure state can be modified to avoid the affected types then the secure state program will not be affected.

All integral Fundamental Data Types with a size less than a word must be changed to an alternative word sized integral Fundamental Data Type. For example, a parameter of char type must be changed to an int type.

If any enumeration types are used in the interface between Secure and Non-Secure state then both Secure and Non-Secure state must strictly conform to the AAPCS32 on enum-size. For armclang, clang and GCC this means compiling with the `-fno-short-enums` option.

Changing the API also changes the ABI, both Secure and Non-secure state must be updated to use the new API.

4.4.3 Inline assembly workaround

If updated tools are unavailable or cannot be used, and the API cannot be changed, then in tools such as armclang, clang and GCC inline assembly can be used to force a zero or sign extension by Secure state.

For a variable V the statement `__asm("" : "+r" (V));` will tell the compiler that the variable in the register is being written to which prevents the compiler from assuming anything about its value.

For example:

```
#include <arm_cmse.h>
#define ARRAY_SIZE (256)

char array[ARRAY_SIZE];

char __attribute__((cmse_nonsecure_entry))
secureFunction(unsigned char index) {
  // Inline assembly output operand tells compiler that index has been
  // written to. Compiler zero-extends to ensure value is within bounds
  // of type.
  __asm("" : "+r"(index));
  // Check is optimized away but value is now within bounds.
  if (index >= ARRAY_SIZE)
    return 0;
  return array[index];
}
```

Using the inline assembly workaround does not require Non-secure state to be rebuilt.

5 Information for Toolchain Developers

This information is for toolchain developers who are implementing C language support for CMSE. It is assumed the reader is familiar with the Armv8-M security model, how C source code maps onto this using CMSE, and how function arguments and return values are passed at machine level following the procedure call standard.

5.1. Is my toolchain affected?

Toolchains are affected if all the following conditions are met:

- The toolchain implements support for Cortex-M CPUs based on the Armv8 architecture or later.
- The toolchain supports generation of secure code following the CMSE standard.
- The toolchain performs no sanitization in Secure state of arguments or return values of less than word size that are passed from non-secure code.

5.2. Toolchain solutions

Affected toolchains should be modified to sanitize arguments and return values that are passed from Non-secure to Secure state where their size is less than a word (see Affected types). The critical change is to sanitize affected values in Secure state prior to first use. The recommended approach to sanitizing values is to zero or sign-extend them to word size following the same rules as used elsewhere in the procedure call standard. It might be possible to optimize away sanitization if it can be determined that subsequent use of the value cannot lead to adverse behavior.

6 References

1. Arm v8-M Security Extensions Requirements on Development Tools https://arm-software.github.io/acle/cmse/cmse.html

2. ARMv8-M Secure software guidelines 2.0 https://developer.arm.com/documentation/100720/0200

3. AAPCS32 Procedure Call Standard for the Arm Architecture https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst

4. SEI CERT C Coding Standard INT32-C https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow