# CWE Detail – CWE-1422

## Description

A processor event or prediction may allow incorrect or stale data to
 be forwarded to transient operations, potentially exposing data over a
 covert channel.

## Extended Description

Software may use a variety of techniques to preserve the
 confidentiality of private data that is accessible within the current
 processor context. For example, the memory safety and type safety
 properties of some high-level programming languages help to prevent
 software written in those languages from exposing private data. As a
 second example, software sandboxes may co-locate multiple users'
 software within a single process. The processor's Instruction Set
 Architecture (ISA) may permit one user's software to access another
 user's data (because the software shares the same address space), but
 the sandbox prevents these accesses by using software techniques such
 as bounds checking. If incorrect or stale data can be forwarded (for example, from a
 cache) to transient operations, then the operations'
 microarchitectural side effects may correspond to the data. If an
 attacker can trigger these transient operations and observe their side
 effects through a covert channel, then the attacker may be able to
 infer the data. For example, an attacker process may induce transient
 execution in a victim process that causes the victim to inadvertently
 access and then expose its private data via a covert channel. In the
 software sandbox example, an attacker sandbox may induce transient
 execution in its own code, allowing it to transiently access and
 expose data in a victim sandbox that shares the same address space. Consequently, weaknesses that arise from incorrect/stale data
 forwarding might violate users' expectations of software-based memory
 safety and isolation techniques. If the data forwarding behavior is
 not properly documented by the hardware vendor, this might violate the
 software vendor's expectation of how the hardware should behave.

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2020-0551: A fault, microcode assist, or abort may allow transient
 load operations to forward malicious stale data to dependent
 operations executed by a victim, causing the victim to unintentionally
 access and potentially expose its own data over a covert channel.

**•** CVE-2020-8698: A fast store forwarding predictor may allow store
 operations to forward incorrect data to transient load operations,
 potentially exposing data over a covert channel.

## Modes of Introduction

**•** Architecture and Design: This weakness can be introduced by data speculation techniques,
 or when the processor pipeline is designed to check exception
 conditions concurrently with other operations. This weakness can also
 persist after a CWE-1421 weakness has been mitigated. For example,
 suppose that a processor can forward stale data from a shared
 microarchitectural buffer to dependent transient operations, and
 furthermore suppose that the processor has been patched to flush the
 buffer on context switches. This mitigates the CWE-1421 weakness, but the
 stale-data forwarding behavior may persist as a CWE-1422 weakness unless
 this behavior is also patched.

## Common Consequences

**•** Impact: Read Memory — Notes:

## Potential Mitigations

**•** Architecture and Design: The hardware designer can attempt to prevent transient
 execution from causing observable discrepancies in specific covert
 channels. (Effectiveness: Limited)

**•** Requirements: Processor designers, system software vendors, or other
 agents may choose to restrict the ability of unprivileged software to
 access to high-resolution timers that are commonly used to monitor
 covert channels. (Effectiveness: Defense in Depth)

**•** Requirements: Processor designers may expose instructions or other
 architectural features that allow software to mitigate the effects of
 transient execution, but without disabling predictors. These features
 may also help to limit opportunities for data exposure. (Effectiveness: Moderate)

**•** Requirements: Processor designers may expose registers (for example,
 control registers or model-specific registers) that allow privileged
 and/or user software to disable specific predictors or other hardware
 features that can cause confidential data to be exposed during
 transient execution. (Effectiveness: Limited)

**•** Build and Compilation: Use software techniques (including the use of
 serialization instructions) that are intended to reduce the number of
 instructions that can be executed transiently after a processor event
 or misprediction. (Effectiveness: Incidental)

**•** Build and Compilation: Isolate sandboxes or managed runtimes in separate address
 spaces (separate processes). (Effectiveness: High)

**•** Build and Compilation: Include serialization instructions (for example, LFENCE)
 that prevent processor events or mis-predictions prior to the
 serialization instruction from causing transient execution after the
 serialization instruction. For some weaknesses, a serialization
 instruction can also prevent a processor event or a mis-prediction
 from occurring after the serialization instruction (for example,
 CVE-2018-3639 can allow a processor to predict that a load will not
 depend on an older store; a serialization instruction between the
 store and the load may allow the store to update memory and prevent
 the mis-prediction from happening at all). (Effectiveness: Moderate)

**•** Build and Compilation: Use software techniques that can mitigate the
 consequences of transient execution. For example, address masking can
 be used in some circumstances to prevent out-of-bounds transient
 reads. (Effectiveness: Limited)

**•** Build and Compilation: If the weakness is exposed by a single instruction (or a
 small set of instructions), then the compiler (or JIT, etc.) can be
 configured to prevent the affected instruction(s) from being
 generated, and instead generate an alternate sequence of instructions
 that is not affected by the weakness. (Effectiveness: Limited)

**•** Documentation: If a hardware feature can allow incorrect or stale data
 to be forwarded to transient operations, the hardware designer may opt
 to disclose this behavior in architecture documentation. This
 documentation can inform users about potential consequences and
 effective mitigations. (Effectiveness: High)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** A processor with this weakness will store the value of untrusted\_arg
 (which may be provided by an attacker) to the stack, which is trusted
 memory. Additionally, this store operation will save this value in
 some microarchitectural buffer, for example, the store buffer. In this code sequence, trusted\_ptr is dereferenced while the attacker
 forces a page fault. The faulting load causes the processor to
 mis-speculate by forwarding untrusted\_arg as the (transient) load
 result. The processor then uses untrusted\_arg for the pointer
 dereference. After the fault has been handled and the load has been
 re-issued with the correct argument, secret-dependent information
 stored at the address of trusted\_ptr remains in microarchitectural
 state and can be extracted by an attacker using a vulnerable code
 sequence.

**•** In this example, assume that the parameter idx can only be 0 or 1, and
 assume that idx\_array initially contains all 0s. Observe that the
 assignment to v in line 4 will be array[0], regardless of whether
 idx=0 or idx=1. Now suppose that an attacker repeatedly invokes fn
 with idx=0 to train the store forwarding predictor to predict that the
 store in line 3 will forward the data 4096 to the load idx\_array[idx]
 in line 4. Then, when the attacker invokes fn with idx=1 the predictor
 may cause idx\_array[idx] to transiently produce the incorrect value
 4096, and therefore v will transiently be assigned the value
 array[4096], which otherwise would not have been accessible in line 4. Although this toy example is benign (it doesn't transmit array[4096]
 over a covert channel), an attacker may be able to use similar
 techniques to craft and train malicious code sequences to, for
 example, read data beyond a software sandbox boundary.