# CWE Detail – CWE-1420

## Description

A processor event or prediction may allow incorrect operations (or correct operations with incorrect data) to execute transiently, potentially exposing data over a covert channel.

## Extended Description

When operations execute but do not commit to the processor's  
 architectural state, this is commonly referred to as transient  
 execution. This behavior can occur when the processor mis-predicts an  
 outcome (such as a branch target), or when a processor event (such as  
 an exception or microcode assist, etc.) is handled after younger  
 operations have already executed. Operations that execute transiently  
 may exhibit observable discrepancies (CWE-203) in covert channels  
 [REF-1400] such as data caches. Observable discrepancies of this kind  
 can be detected and analyzed using timing or power analysis  
 techniques, which may allow an attacker to infer information about the  
 operations that executed transiently. For example, the attacker may be  
 able to infer confidential data that was accessed or used by those  
 operations. Transient execution weaknesses may be exploited using one of two  
 methods. In the first method, the attacker generates a code sequence  
 that exposes data through a covert channel when it is executed  
 transiently (the attacker must also be able to trigger transient  
 execution). Some transient execution weaknesses can only expose data  
 that is accessible within the attacker's processor context. For  
 example, an attacker executing code in a software sandbox may be able  
 to use a transient execution weakness to expose data within the same  
 address space, but outside of the attacker's sandbox. Other transient  
 execution weaknesses can expose data that is architecturally  
 inaccessible, that is, data protected by hardware-enforced boundaries  
 such as page tables or privilege rings. These weaknesses are the  
 subject of CWE-1421. In the second exploitation method, the attacker first identifies a  
 code sequence in a victim program that, when executed transiently, can  
 expose data that is architecturally accessible within the victim's  
 processor context. For instance, the attacker may search the victim  
 program for code sequences that resemble a bounds-check bypass  
 sequence (see Demonstrative Example 1). If the attacker can trigger a  
 mis-prediction of the conditional branch and influence the index of  
 the out-of-bounds array access, then the attacker may be able to infer  
 the value of out-of-bounds data by monitoring observable discrepancies  
 in a covert channel.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2017-5753: Microarchitectural conditional branch predictors may allow operations to execute transiently after a misprediction, potentially exposing data over a covert channel.

**•** CVE-2021-0089: A machine clear triggered by self-modifying code may allow incorrect operations to execute transiently, potentially exposing data over a covert channel.

**•** CVE-2022-0002: Microarchitectural indirect branch predictors may allow incorrect operations to execute transiently after a misprediction, potentially exposing data over a covert channel.

## Modes of Introduction

**•** Architecture and Design: This weakness can be introduced when a computing unit (such as a CPU, GPU, accelerator, or any other processor) uses out-of-order execution, speculation, or any other microarchitectural feature that can allow microarchitectural operations to execute without committing to architectural state.

**•** Implementation: This weakness can be introduced when sandboxes or managed runtimes are not properly isolated by using hardware-enforced boundaries. Developers of sandbox or managed runtime software should exercise caution when relying on software techniques (such as bounds checking) to prevent code in one sandbox from accessing confidential data in another sandbox. For example, an attacker sandbox may be able to trigger a processor event or mis-prediction in a manner that allows it to transiently read a victim sandbox's private data.

## 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 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)

**•** 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)

**•** Build and Compilation: Isolate sandboxes or managed runtimes in separate address spaces (separate processes). For examples, see [REF-1421]. (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 prediction from happening at all). (Effectiveness: Moderate)

**•** Build and Compilation: Use control-flow integrity (CFI) techniques to constrain the behavior of instructions that redirect the instruction pointer, such as indirect branch instructions. (Effectiveness: Moderate)

**•** 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. One prominent example of this mitigation is retpoline ([REF-1414]). (Effectiveness: Limited)

**•** 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: 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)

**•** Documentation: If a hardware feature can allow incorrect operations (or correct operations with incorrect data) to execute transiently, 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

**•** However, if this code executes on a processor that performs  
 conditional branch prediction the outcome of the if statement could be  
 mis-predicted and the access on the next line will occur with a value  
 of x that can point to an out-of-bounds location (within the program's  
 memory). Even though the processor does not commit the architectural effects of  
 the mis-predicted branch, the memory accesses alter data cache state,  
 which is not rolled back after the branch is resolved. The cache state  
 can reveal array1[x] thereby providing a mechanism to recover the data  
 value located at address array1 + x.

**•** N/A