# CWE Detail – CWE-362

## Description

The product contains a concurrent code sequence that requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence operating concurrently.

## Extended Description

A race condition occurs within concurrent environments, and it is effectively a property of a code sequence. Depending on the context, a code sequence may be in the form of a function call, a small number of instructions, a series of program invocations, etc. A race condition violates these properties, which are closely related: Exclusivity - the code sequence is given exclusive access to the shared resource, i.e., no other code sequence can modify properties of the shared resource before the original sequence has completed execution. Atomicity - the code sequence is behaviorally atomic, i.e., no other thread or process can concurrently execute the same sequence of instructions (or a subset) against the same resource. A race condition exists when an "interfering code sequence" can still access the shared resource, violating exclusivity. The interfering code sequence could be "trusted" or "untrusted." A trusted interfering code sequence occurs within the product; it cannot be modified by the attacker, and it can only be invoked indirectly. An untrusted interfering code sequence can be authored directly by the attacker, and typically it is external to the vulnerable product.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2022-29527: Go application for cloud management creates a world-writable sudoers file that allows local attackers to inject sudo rules and escalate privileges to root by winning a race condition.

**•** CVE-2021-1782: Chain: improper locking (CWE-667) leads to race condition (CWE-362), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-0920: Chain: mobile platform race condition (CWE-362) leading to use-after-free (CWE-416), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-6819: Chain: race condition (CWE-362) leads to use-after-free (CWE-416), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2019-18827: chain: JTAG interface is not disabled (CWE-1191) during ROM code execution, introducing a race condition (CWE-362) to extract encryption keys

**•** CVE-2019-1161: Chain: race condition (CWE-362) in anti-malware product allows deletion of files by creating a junction (CWE-1386) and using hard links during the time window in which a temporary file is created and deleted.

**•** CVE-2015-1743: TOCTOU in sandbox process allows installation of untrusted browser add-ons by replacing a file after it has been verified, but before it is executed

**•** CVE-2014-8273: Chain: chipset has a race condition (CWE-362) between when an interrupt handler detects an attempt to write-enable the BIOS (in violation of the lock bit), and when the handler resets the write-enable bit back to 0, allowing attackers to issue BIOS writes during the timing window [REF-1237].

**•** CVE-2008-5044: Race condition leading to a crash by calling a hook removal procedure while other activities are occurring at the same time.

**•** CVE-2008-2958: chain: time-of-check time-of-use (TOCTOU) race condition in program allows bypass of protection mechanism that was designed to prevent symlink attacks.

**•** CVE-2008-1570: chain: time-of-check time-of-use (TOCTOU) race condition in program allows bypass of protection mechanism that was designed to prevent symlink attacks.

**•** CVE-2008-0058: Unsynchronized caching operation enables a race condition that causes messages to be sent to a deallocated object.

**•** CVE-2008-0379: Race condition during initialization triggers a buffer overflow.

**•** CVE-2007-6599: Daemon crash by quickly performing operations and undoing them, which eventually leads to an operation that does not acquire a lock.

**•** CVE-2007-6180: chain: race condition triggers NULL pointer dereference

**•** CVE-2007-5794: Race condition in library function could cause data to be sent to the wrong process.

**•** CVE-2007-3970: Race condition in file parser leads to heap corruption.

**•** CVE-2008-5021: chain: race condition allows attacker to access an object while it is still being initialized, causing software to access uninitialized memory.

**•** CVE-2009-4895: chain: race condition for an argument value, possibly resulting in NULL dereference

**•** CVE-2009-3547: chain: race condition might allow resource to be released before operating on it, leading to NULL dereference

**•** CVE-2006-5051: Chain: Signal handler contains too much functionality (CWE-828), introducing a race condition (CWE-362) that leads to a double free (CWE-415).

## Related Attack Patterns (CAPEC)

* CAPEC-26
* CAPEC-29

## Modes of Introduction

**•** Architecture and Design: N/A

**•** Implementation: Programmers may assume that certain code sequences execute too quickly to be affected by an interfering code sequence; when they are not, this violates atomicity. For example, the single "x++" statement may appear atomic at the code layer, but it is actually non-atomic at the instruction layer, since it involves a read (the original value of x), followed by a computation (x+1), followed by a write (save the result to x).

## Common Consequences

**•** Impact: DoS: Resource Consumption (CPU), DoS: Resource Consumption (Memory), DoS: Resource Consumption (Other) — Notes: When a race condition makes it possible to bypass a resource cleanup routine or trigger multiple initialization routines, it may lead to resource exhaustion.

**•** Impact: DoS: Crash, Exit, or Restart, DoS: Instability — Notes: When a race condition allows multiple control flows to access a resource simultaneously, it might lead the product(s) into unexpected states, possibly resulting in a crash.

**•** Impact: Read Files or Directories, Read Application Data — Notes: When a race condition is combined with predictable resource names and loose permissions, it may be possible for an attacker to overwrite or access confidential data (CWE-59).

**•** Impact: Execute Unauthorized Code or Commands, Gain Privileges or Assume Identity, Bypass Protection Mechanism — Notes: This can have security implications when the expected synchronization is in security-critical code, such as recording whether a user is authenticated or modifying important state information that should not be influenced by an outsider.

## Potential Mitigations

**•** Architecture and Design: In languages that support it, use synchronization primitives. Only wrap these around critical code to minimize the impact on performance. (Effectiveness: N/A)

**•** Architecture and Design: Use thread-safe capabilities such as the data access abstraction in Spring. (Effectiveness: N/A)

**•** Architecture and Design: Minimize the usage of shared resources in order to remove as much complexity as possible from the control flow and to reduce the likelihood of unexpected conditions occurring. Additionally, this will minimize the amount of synchronization necessary and may even help to reduce the likelihood of a denial of service where an attacker may be able to repeatedly trigger a critical section (CWE-400). (Effectiveness: N/A)

**•** Implementation: When using multithreading and operating on shared variables, only use thread-safe functions. (Effectiveness: N/A)

**•** Implementation: Use atomic operations on shared variables. Be wary of innocent-looking constructs such as "x++". This may appear atomic at the code layer, but it is actually non-atomic at the instruction layer, since it involves a read, followed by a computation, followed by a write. (Effectiveness: N/A)

**•** Implementation: Use a mutex if available, but be sure to avoid related weaknesses such as CWE-412. (Effectiveness: N/A)

**•** Implementation: Avoid double-checked locking (CWE-609) and other implementation errors that arise when trying to avoid the overhead of synchronization. (Effectiveness: N/A)

**•** Implementation: Disable interrupts or signals over critical parts of the code, but also make sure that the code does not go into a large or infinite loop. (Effectiveness: N/A)

**•** Implementation: Use the volatile type modifier for critical variables to avoid unexpected compiler optimization or reordering. This does not necessarily solve the synchronization problem, but it can help. (Effectiveness: N/A)

**•** Architecture and Design: Run your code using the lowest privileges that are required to accomplish the necessary tasks [REF-76]. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations. (Effectiveness: N/A)

## Applicable Platforms

**•** C (Class: None, Prevalence: Sometimes)

**•** C++ (Class: None, Prevalence: Sometimes)

**•** Java (Class: None, Prevalence: Sometimes)

## Demonstrative Examples

**•** A race condition could occur between the calls to GetBalanceFromDatabase() and SendNewBalanceToDatabase().

**•** However, the code does not check the value returned by pthread\_mutex\_lock() for errors. If pthread\_mutex\_lock() cannot acquire the mutex for any reason, the function may introduce a race condition into the program and result in undefined behavior.

**•** Suppose the interconnect fabric does not prioritize such "update" packets over other general traffic packets. This introduces a race condition. If an attacker can flood the target with enough messages so that some of those attack packets reach the target before the new access ranges gets updated, then the attacker can leverage this scenario.

## Notes

**•** Maintenance: The relationship between race conditions and synchronization problems (CWE-662) needs to be further developed. They are not necessarily two perspectives of the same core concept, since synchronization is only one technique for avoiding race conditions, and synchronization can be used for other purposes besides race condition prevention.

**•** Research Gap: Race conditions in web applications are under-studied and probably under-reported. However, in 2008 there has been growing interest in this area.

**•** Research Gap: Much of the focus of race condition research has been in Time-of-check Time-of-use (TOCTOU) variants (CWE-367), but many race conditions are related to synchronization problems that do not necessarily require a time-of-check.

**•** Research Gap: From a classification/taxonomy perspective, the relationships between concurrency and program state need closer investigation and may be useful in organizing related issues.