# CWE Detail – CWE-1265

## Description

During execution of non-reentrant code, the product performs a call that unintentionally produces a nested invocation of the non-reentrant code.

## Extended Description

In a complex product, a single function call may lead to many different possible code paths, some of which may involve deeply nested calls. It may be difficult to foresee all possible code paths that could emanate from a given function call. In some systems, an external actor can manipulate inputs to the system and thereby achieve a wide range of possible control flows. This is frequently a concern in products that execute scripts from untrusted sources. Examples of such products are web browsers and PDF readers. A weakness is present when one of the possible code paths resulting from a function call alters program state that the original caller assumes to be unchanged during the call.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2014-1772: In this vulnerability, by registering a malicious onerror handler, an adversary can produce unexpected re-entrance of a CDOMRange object. [REF-1098]

**•** CVE-2018-8174: This CVE covers several vulnerable scenarios enabled by abuse of the Class\_Terminate feature in Microsoft VBScript. In one scenario, Class\_Terminate is used to produce an undesirable re-entrance of ScriptingDictionary during execution of that object's destructor. In another scenario, a vulnerable condition results from a recursive entrance of a property setter method. This recursive invocation produces a second, spurious call to the Release method of a reference-counted object, causing a UAF when that object is freed prematurely. This vulnerability pattern has been popularized as "Double Kill". [REF-1099] (KEV)

## Related Attack Patterns (CAPEC)

* CAPEC-74

## Common Consequences

**•** Impact: Unexpected State — Notes: Exploitation of this weakness can leave the application in an unexpected state and cause variables to be reassigned before the first invocation has completed. This may eventually result in memory corruption or unexpected code execution.

## Potential Mitigations

**•** Architecture and Design: When architecting a system that will execute untrusted code in response to events, consider executing the untrusted event handlers asynchronously (asynchronous message passing) as opposed to executing them synchronously at the time each event fires. The untrusted code should execute at the start of the next iteration of the thread's message loop. In this way, calls into non-reentrant code are strictly serialized, so that each operation completes fully before the next operation begins. Special attention must be paid to all places where type coercion may result in script execution. Performing all needed coercions at the very beginning of an operation can help reduce the chance of operations executing at unexpected junctures. (Effectiveness: High)

**•** Implementation: Make sure the code (e.g., function or class) in question is reentrant by not leveraging non-local data, not modifying its own code, and not calling other non-reentrant code. (Effectiveness: High)

## Applicable Platforms

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

## Demonstrative Examples

**•** Looking closer at this example, Widget::click() calls backgroundImage->click(), which in turn calls scriptEngine->fireOnImageClick(). The code within fireOnImageClick() invokes the appropriate script handler routine as defined by the document being rendered. In this scenario this script routine is supplied by an adversary and this malicious script makes a call to Widget::changeBackgroundImage(), deleting the Image object pointed to by backgroundImage. When control returns to Image::click, the function's backgroundImage "this" pointer (which is the former value of backgroundImage) is a dangling pointer. The root of this weakness is that while one operation on Widget (click) is in the midst of executing, a second operation on the Widget object may be invoked (in this case, the second invocation is a call to different method, namely changeBackgroundImage) that modifies the non-local variable.

**•** The expected order of operations is a call to Request::setup(), followed by a call to Request::send(). Request::send() calls scriptEngine->coerceToString(\_data) to coerce a script-provided parameter into a string. This operation may produce script execution. For example, if the script language is ECMAScript, arbitrary script execution may result if \_data is an adversary-supplied ECMAScript object having a custom toString method. If the adversary's script makes a new call to Request::setup, then when control returns to Request::send, the field uri and the local variable credentials will no longer be consistent with one another. As a result, credentials for one resource will be shared improperly with a different resource. The root of this weakness is that while one operation on Request (send) is in the midst of executing, a second operation may be invoked (setup).