# CWE Detail – CWE-829

## Description

The product imports, requires, or includes executable functionality (such as a library) from a source that is outside of the intended control sphere.

## Extended Description

When including third-party functionality, such as a web widget, library, or other source of functionality, the product must effectively trust that functionality. Without sufficient protection mechanisms, the functionality could be malicious in nature (either by coming from an untrusted source, being spoofed, or being modified in transit from a trusted source). The functionality might also contain its own weaknesses, or grant access to additional functionality and state information that should be kept private to the base system, such as system state information, sensitive application data, or the DOM of a web application. This might lead to many different consequences depending on the included functionality, but some examples include injection of malware, information exposure by granting excessive privileges or permissions to the untrusted functionality, DOM-based XSS vulnerabilities, stealing user's cookies, or open redirect to malware (CWE-601).

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2010-2076: Product does not properly reject DTDs in SOAP messages, which allows remote attackers to read arbitrary files, send HTTP requests to intranet servers, or cause a denial of service.

**•** CVE-2004-0285: Modification of assumed-immutable configuration variable in include file allows file inclusion via direct request.

**•** CVE-2004-0030: Modification of assumed-immutable configuration variable in include file allows file inclusion via direct request.

**•** CVE-2004-0068: Modification of assumed-immutable configuration variable in include file allows file inclusion via direct request.

**•** CVE-2005-2157: Modification of assumed-immutable configuration variable in include file allows file inclusion via direct request.

**•** CVE-2005-2162: Modification of assumed-immutable configuration variable in include file allows file inclusion via direct request.

**•** CVE-2005-2198: Modification of assumed-immutable configuration variable in include file allows file inclusion via direct request.

**•** CVE-2004-0128: Modification of assumed-immutable variable in configuration script leads to file inclusion.

**•** CVE-2005-1864: PHP file inclusion.

**•** CVE-2005-1869: PHP file inclusion.

**•** CVE-2005-1870: PHP file inclusion.

**•** CVE-2005-2154: PHP local file inclusion.

**•** CVE-2002-1704: PHP remote file include.

**•** CVE-2002-1707: PHP remote file include.

**•** CVE-2005-1964: PHP remote file include.

**•** CVE-2005-1681: PHP remote file include.

**•** CVE-2005-2086: PHP remote file include.

**•** CVE-2004-0127: Directory traversal vulnerability in PHP include statement.

**•** CVE-2005-1971: Directory traversal vulnerability in PHP include statement.

**•** CVE-2005-3335: PHP file inclusion issue, both remote and local; local include uses ".." and "%00" characters as a manipulation, but many remote file inclusion issues probably have this vector.

## Related Attack Patterns (CAPEC)

* CAPEC-175
* CAPEC-201
* CAPEC-228
* CAPEC-251
* CAPEC-252
* CAPEC-253
* CAPEC-263
* CAPEC-538
* CAPEC-549
* CAPEC-640
* CAPEC-660
* CAPEC-695
* CAPEC-698

## Attack TTPs

**•** T1195.001: Compromise Software Dependencies and Development Tools (Tactics: initial-access)

**•** T1505.005: Terminal Services DLL (Tactics: persistence)

**•** T1176: Software Extensions (Tactics: persistence)

**•** T1055: Process Injection (Tactics: defense-evasion, privilege-escalation)

**•** T1620: Reflective Code Loading (Tactics: defense-evasion)

**•** T1574.006: Dynamic Linker Hijacking (Tactics: persistence, privilege-escalation, defense-evasion)

**•** T1574.013: KernelCallbackTable (Tactics: persistence, privilege-escalation, defense-evasion)

**•** T1505.004: IIS Components (Tactics: persistence)

## Modes of Introduction

**•** Implementation: REALIZATION: This weakness is caused during implementation of an architectural security tactic.

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands — Notes: An attacker could insert malicious functionality into the program by causing the program to download code that the attacker has placed into the untrusted control sphere, such as a malicious web site.

## Potential Mitigations

**•** Architecture and Design: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. (Effectiveness: N/A)

**•** Architecture and Design: When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap [REF-45] provide this capability. (Effectiveness: N/A)

**•** Architecture and Design: For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. (Effectiveness: N/A)

**•** Architecture and Design: Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by the software. OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows the software to specify restrictions on file operations. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails. (Effectiveness: Limited)

**•** 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)

**•** Implementation: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When validating filenames, use stringent allowlists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a list of allowable file extensions, which will help to avoid CWE-434. Do not rely exclusively on a filtering mechanism that removes potentially dangerous characters. This is equivalent to a denylist, which may be incomplete (CWE-184). For example, filtering "/" is insufficient protection if the filesystem also supports the use of "\" as a directory separator. Another possible error could occur when the filtering is applied in a way that still produces dangerous data (CWE-182). For example, if "../" sequences are removed from the ".../...//" string in a sequential fashion, two instances of "../" would be removed from the original string, but the remaining characters would still form the "../" string. (Effectiveness: High)

**•** Architecture and Design: Store library, include, and utility files outside of the web document root, if possible. Otherwise, store them in a separate directory and use the web server's access control capabilities to prevent attackers from directly requesting them. One common practice is to define a fixed constant in each calling program, then check for the existence of the constant in the library/include file; if the constant does not exist, then the file was directly requested, and it can exit immediately. This significantly reduces the chance of an attacker being able to bypass any protection mechanisms that are in the base program but not in the include files. It will also reduce the attack surface. (Effectiveness: N/A)

**•** Architecture and Design: Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls. Many file inclusion problems occur because the programmer assumed that certain inputs could not be modified, especially for cookies and URL components. (Effectiveness: N/A)

**•** Operation: Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth. (Effectiveness: Moderate)

## Demonstrative Examples

**•** This webpage is now only as secure as the external domain it is including functionality from. If an attacker compromised the external domain and could add malicious scripts to the weatherwidget.js file, the attacker would have complete control, as seen in any XSS weakness (CWE-79).