# CWE Detail – CWE-22

## Description

The product uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the product does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.

## Extended Description

Many file operations are intended to take place within a restricted directory. By using special elements such as ".." and "/" separators, attackers can escape outside of the restricted location to access files or directories that are elsewhere on the system. One of the most common special elements is the "../" sequence, which in most modern operating systems is interpreted as the parent directory of the current location. This is referred to as relative path traversal. Path traversal also covers the use of absolute pathnames such as "/usr/local/bin" to access unexpected files. This is referred to as absolute path traversal.

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2024-37032: Large language model (LLM) management tool does not
 validate the format of a digest value (CWE-1287) from a
 private, untrusted model registry, enabling relative
 path traversal (CWE-23), a.k.a. Probllama

**•** CVE-2024-4315: Chain: API for text generation using Large Language Models (LLMs) does
 not include the "\" Windows folder separator in its denylist (CWE-184)
 when attempting to prevent Local File Inclusion via path traversal
 (CWE-22), allowing deletion of arbitrary files on Windows systems.

**•** CVE-2022-45918: Chain: a learning management tool debugger uses external input to locate previous session logs (CWE-73) and does not properly validate the given path (CWE-20), allowing for filesystem path traversal using "../" sequences (CWE-24)

**•** CVE-2019-20916: Python package manager does not correctly restrict the filename specified in a Content-Disposition header, allowing arbitrary file read using path traversal sequences such as "../"

**•** CVE-2022-31503: Python package constructs filenames using an unsafe os.path.join call on untrusted input, allowing absolute path traversal because os.path.join resets the pathname to an absolute path that is specified as part of the input.

**•** CVE-2022-24877: directory traversal in Go-based Kubernetes operator app allows accessing data from the controller's pod file system via ../ sequences in a yaml file

**•** CVE-2021-21972: Chain: Cloud computing virtualization platform does not require authentication for upload of a tar format file (CWE-306), then uses .. path traversal sequences (CWE-23) in the file to access unexpected files, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-4053: a Kubernetes package manager written in Go allows malicious plugins to inject path traversal sequences into a plugin archive ("Zip slip") to copy a file outside the intended directory

**•** CVE-2020-3452: Chain: security product has improper input validation (CWE-20) leading to directory traversal (CWE-22), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2019-10743: Go-based archive library allows extraction of files to locations outside of the target folder with "../" path traversal sequences in filenames in a zip file, aka "Zip Slip"

**•** CVE-2010-0467: Newsletter module allows reading arbitrary files using "../" sequences.

**•** CVE-2006-7079: Chain: PHP app uses extract for register\_globals compatibility layer (CWE-621), enabling path traversal (CWE-22)

**•** CVE-2009-4194: FTP server allows deletion of arbitrary files using ".." in the DELE command.

**•** CVE-2009-4053: FTP server allows creation of arbitrary directories using ".." in the MKD command.

**•** CVE-2009-0244: FTP service for a Bluetooth device allows listing of directories, and creation or reading of files using ".." sequences.

**•** CVE-2009-4013: Software package maintenance program allows overwriting arbitrary files using "../" sequences.

**•** CVE-2009-4449: Bulletin board allows attackers to determine the existence of files using the avatar.

**•** CVE-2009-4581: PHP program allows arbitrary code execution using ".." in filenames that are fed to the include() function.

**•** CVE-2010-0012: Overwrite of files using a .. in a Torrent file.

**•** CVE-2010-0013: Chat program allows overwriting files using a custom smiley request.

**•** CVE-2008-5748: Chain: external control of values for user's desired language and theme enables path traversal.

**•** CVE-2009-1936: Chain: library file sends a redirect if it is directly requested but continues to execute, allowing remote file inclusion and path traversal.

## Related Attack Patterns (CAPEC)

* CAPEC-126
* CAPEC-64
* CAPEC-76
* CAPEC-78
* CAPEC-79

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands — Notes: The attacker may be able to create or overwrite critical files that are used to execute code, such as programs or libraries.

**•** Impact: Modify Files or Directories — Notes: The attacker may be able to overwrite or create critical files, such as programs, libraries, or important data. If the targeted file is used for a security mechanism, then the attacker may be able to bypass that mechanism. For example, appending a new account at the end of a password file may allow an attacker to bypass authentication.

**•** Impact: Read Files or Directories — Notes: The attacker may be able read the contents of unexpected files and expose sensitive data. If the targeted file is used for a security mechanism, then the attacker may be able to bypass that mechanism. For example, by reading a password file, the attacker could conduct brute force password guessing attacks in order to break into an account on the system.

**•** Impact: DoS: Crash, Exit, or Restart — Notes: The attacker may be able to overwrite, delete, or corrupt unexpected critical files such as programs, libraries, or important data. This may prevent the product from working at all and in the case of protection mechanisms such as authentication, it has the potential to lock out product users.

## Potential Mitigations

**•** 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: 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)

**•** Implementation: Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180). Make sure that the application does not decode the same input twice (CWE-174). Such errors could be used to bypass allowlist validation schemes by introducing dangerous inputs after they have been checked. Use a built-in path canonicalization function (such as realpath() in C) that produces the canonical version of the pathname, which effectively removes ".." sequences and symbolic links (CWE-23, CWE-59). This includes: realpath() in C getCanonicalPath() in Java GetFullPath() in ASP.NET realpath() or abs\_path() in Perl realpath() in PHP (Effectiveness: N/A)

**•** 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)

**•** 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)

**•** 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)

**•** 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-185] provide this capability. (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: 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)

**•** Implementation: Ensure that error messages only contain minimal details that are useful to the intended audience and no one else. The messages need to strike the balance between being too cryptic (which can confuse users) or being too detailed (which may reveal more than intended). The messages should not reveal the methods that were used to determine the error. Attackers can use detailed information to refine or optimize their original attack, thereby increasing their chances of success. If errors must be captured in some detail, record them in log messages, but consider what could occur if the log messages can be viewed by attackers. Highly sensitive information such as passwords should never be saved to log files. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a user account exists or not. In the context of path traversal, error messages which disclose path information can help attackers craft the appropriate attack strings to move through the file system hierarchy. (Effectiveness: N/A)

**•** Operation: When using PHP, configure the application so that it does not use register\_globals. During implementation, develop the application so that it does not rely on this feature, but be wary of implementing a register\_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues. (Effectiveness: N/A)

## Applicable Platforms

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

## Demonstrative Examples

**•** While the programmer intends to access files such as "/users/cwe/profiles/alice" or "/users/cwe/profiles/bob", there is no verification of the incoming user parameter. An attacker could provide a string such as:

**•** However, the path is not validated or modified to prevent it from containing relative or absolute path sequences before creating the File object. This allows anyone who can control the system property to determine what file is used. Ideally, the path should be resolved relative to some kind of application or user home directory.

**•** Since the regular expression does not have the /g global match modifier, it only removes the first instance of "../" it comes across. So an input value such as:

**•** An attacker could provide an input such as this:

**•** When submitted the Java servlet's doPost method will receive the request, extract the name of the file from the Http request header, read the file contents from the request and output the file to the local upload directory.

**•** However, if the user supplies an absolute path, the os.path.join() function will discard the path to the current working directory and use only the absolute path provided. For example, if the current working directory is /home/user/documents, but the user inputs /etc/passwd, os.path.join() will use only /etc/passwd, as it is considered an absolute path. In the above scenario, this would cause the script to access and read the /etc/passwd file.

## Notes

**•** Other: In many programming languages, the injection of a null byte (the 0 or NUL) may allow an attacker to truncate a generated filename to apply to a wider range of files. For example, the product may add ".txt" to any pathname, thus limiting the attacker to text files, but a null injection may effectively remove this restriction.

**•** Relationship: Pathname equivalence can be regarded as a type of canonicalization error.

**•** Relationship: Some pathname equivalence issues are not directly related to directory traversal, rather are used to bypass security-relevant checks for whether a file/directory can be accessed by the attacker (e.g. a trailing "/" on a filename could bypass access rules that don't expect a trailing /, causing a server to provide the file when it normally would not).

**•** Terminology: Like other weaknesses, terminology is often based on the types of manipulations used, instead of the underlying weaknesses. Some people use "directory traversal" only to refer to the injection of ".." and equivalent sequences whose specific meaning is to traverse directories. Other variants like "absolute pathname" and "drive letter" have the \*effect\* of directory traversal, but some people may not call it such, since it doesn't involve ".." or equivalent.

**•** Research Gap: Many variants of path traversal attacks are probably under-studied with respect to root cause. CWE-790 and CWE-182 begin to cover part of this gap.

**•** Research Gap: Incomplete diagnosis or reporting of vulnerabilities can make it difficult to know which variant is affected. For example, a researcher might say that "..\" is vulnerable, but not test "../" which may also be vulnerable. Any combination of directory separators ("/", "\", etc.) and numbers of "." (e.g. "....") can produce unique variants; for example, the "//../" variant is not listed (CVE-2004-0325). See this entry's children and lower-level descendants.