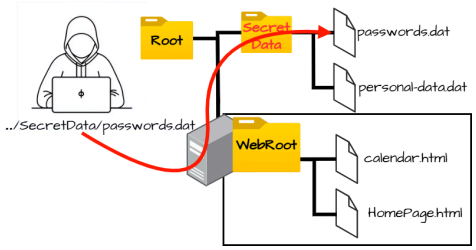
CWE-23: Relative Path Traversal

Weakness ID: 23
Vulnerability Mapping: ALLOWED
Abstraction: Base

View customized information: Conceptual | Operational | Mapping Friendly | **Complete** | Custom

Description

The product uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory.



Alternate Terms

Zip Slip	"Zip slip" is an attack that uses file archives (e.g., ZIP, tar, rar, etc.) that contain filenames with path traversal sequences that cause the files to be written outside of the directory under which the archive is expected to be extracted [REF-1282]. It is most commonly used for relative path traversal (CWE-23) and link following (CWE-59).

Common Consequences

Impact	Details
Execute Unauthorized Code or Commands	**Scope: Integrity, Confidentiality, Availability** The attacker may be able to create or overwrite critical files that are used to execute code, such as programs or libraries.
Modify Files or Directories	**Scope: Integrity** 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.
Read Files or Directories	**Scope: Confidentiality** The attacker may be able read the contents of unexpected files and expose sensitive data by traversing the file system to access files or directories that are outside of the restricted directory. 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.
DoS: Crash, Exit, or Restart	**Scope: Availability** 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 a protection mechanisms such as authentication, it has the potential to lockout every user of the product.

Potential Mitigations

Phase(s)	Mitigation
Implementation	**Strategy:** ***Input Validation***

	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.
Implementation	**Strategy: *Input Validation*** 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
Operation	**Strategy: *Firewall*** 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 [REF-1481]. **Effectiveness: Moderate** **Note:** An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Relationships

Relevant to the view "Research Concepts" (View-1000)

Nature	Type	ID	Name
ChildOf	B	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
ParentOf	V	24	Path Traversal: '../filedir'
ParentOf	V	25	Path Traversal: '/../filedir'
ParentOf	V	26	Path Traversal: '/dir/../filename'
ParentOf	V	27	Path Traversal: 'dir/../../filename'
ParentOf	V	28	Path Traversal: '..\filedir'
ParentOf	V	29	Path Traversal: '\..\filename'
ParentOf	V	30	Path Traversal: '\dir\..\filename'
ParentOf	V	31	Path Traversal: 'dir\..\..\filename'
ParentOf	V	32	Path Traversal: '...' (Triple Dot)
ParentOf	V	33	Path Traversal: '....' (Multiple Dot)
ParentOf	V	34	Path Traversal: '....//'
ParentOf	V	35	Path Traversal: '.../...//'

Relevant to the view "CISQ Quality Measures (2020)" (View-1305)

Nature	Type	ID	Name
ChildOf	B	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Relevant to the view "CISQ Data Protection Measures" (View-1340)

Nature	Type	ID	Name
ChildOf	B	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Modes Of Introduction

Phase	Note
Implementation	

Applicable Platforms

Languages	Class: Not Language-Specific *(Undetermined Prevalence)*
Technologies	Class: Not Technology-Specific *(Undetermined Prevalence)* Class: Web Based *(Undetermined Prevalence)* AI/ML *(Undetermined Prevalence)*

Demonstrative Examples

Example 1

The following URLs are vulnerable to this attack:

Example Language: **Other** *(bad code)*

```
http://example.com/get-files.jsp?file=report.pdf
http://example.com/get-page.php?home=aaa.html
http://example.com/some-page.asp?page=index.html
```

A simple way to execute this attack is like this:

Example Language: **Other** *(attack code)*

```
http://example.com/get-files?file=../../../../somedir/somefile
http://example.com/../../../../etc/shadow
http://example.com/get-files?file=../../../../etc/passwd
```

Example 2

The following code could be for a social networking application in which each user's profile information is stored in a separate file. All files are stored in a single directory.

Example Language: **Perl** *(bad code)*

```
my $dataPath = "/users/cwe/profiles";
my $username = param("user");
my $profilePath = $dataPath . "/" . $username;

open(my $fh, "<", $profilePath) || ExitError("profile read error: $profilePath");
print "<ul>\n";
while (<$fh>) {
   print "<li>$_</li>\n";
}
print "</ul>\n";
```

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:

(attack code)

```
../../../etc/passwd
```

The program would generate a profile pathname like this:

(result)

```
/users/cwe/profiles/../../../etc/passwd
```

When the file is opened, the operating system resolves the "../" during path canonicalization and actually accesses this file:

(result)

```
/etc/passwd
```

As a result, the attacker could read the entire text of the password file.

Notice how this code also contains an error message information leak (CWE-209) if the user parameter does not produce a file that exists: the full pathname is provided. Because of the lack of output encoding of the file that is retrieved, there might also be a cross-site scripting problem (CWE-79) if profile contains any HTML, but other code would need to be examined.

Example 3

The following code demonstrates the unrestricted upload of a file with a Java servlet and a path traversal vulnerability. The action attribute of an HTML form is sending the upload file request to the Java servlet.

Example Language: **HTML** *(good code)*

```
<form action="FileUploadServlet" method="post" enctype="multipart/form-data">

Choose a file to upload:
<input type="file" name="filename"/>
<br/>
<input type="submit" name="submit" value="Submit"/>

</form>
```

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.

Example Language: **Java** *(bad code)*

```
public class FileUploadServlet extends HttpServlet {
  ...

  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
  IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String contentType = request.getContentType();

    // the starting position of the boundary header
    int ind = contentType.indexOf("boundary=");
    String boundary = contentType.substring(ind+9);

    String pLine = new String();
    String uploadLocation = new String(UPLOAD_DIRECTORY_STRING); //Constant value

    // verify that content type is multipart form data
    if (contentType != null && contentType.indexOf("multipart/form-data") != -1) {
      // extract the filename from the Http header
      BufferedReader br = new BufferedReader(new InputStreamReader(request.getInputStream()));
      ...
      pLine = br.readLine();
      String filename = pLine.substring(pLine.lastIndexOf("\\"), pLine.lastIndexOf("\""));
      ...

      // output the file to the local upload directory
      try {
        BufferedWriter bw = new BufferedWriter(new FileWriter(uploadLocation+filename, true));
        for (String line; (line=br.readLine())!=null; ) {
          if (line.indexOf(boundary) == -1) {
            bw.write(line);
            bw.newLine();
            bw.flush();
          }
```

```
            } //end of for loop
            bw.close();


          } catch (IOException ex) {...}
          // output successful upload response HTML page
        }
        // output unsuccessful upload response HTML page
        else
        {...}
    }
        ...
}
```

This code does not perform a check on the type of the file being uploaded (CWE-434). This could allow an attacker to upload any executable file or other file with malicious code.

Additionally, the creation of the BufferedWriter object is subject to relative path traversal (CWE-23). Since the code does not check the filename that is provided in the header, an attacker can use "../" sequences to write to files outside of the intended directory. Depending on the executing environment, the attacker may be able to specify arbitrary files to write to, leading to a wide variety of consequences, from code execution, XSS (CWE-79), or system crash.

Selected Observed Examples

Note: this is a curated list of examples for users to understand the variety of ways in which this weakness can be introduced. It is not a complete list of all CVEs that are related to this CWE entry.

Reference	Description
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-2024-0520	Product for managing datasets for AI model training and evaluation allows both relative (CWE-23) and absolute (CWE-36) path traversal to overwrite files via the Content-Disposition header
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-2022-24877	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-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-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.
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-2002-0298	Server allows remote attackers to cause a denial of service via certain HTTP GET requests containing a %2e%2e (encoded dot-dot), several "/../" sequences, or several "../" in a URI.
CVE-2002-0661	"\" not in denylist for web server, allowing path traversal attacks when the server is run in Windows and other OSes.
CVE-2002-0946	Arbitrary files may be read files via ..\ (dot dot) sequences in an HTTP request.
CVE-2002-1042	Directory traversal vulnerability in search engine for web server allows remote attackers to read arbitrary files via "..\" sequences in queries.
CVE-2002-1209	Directory traversal vulnerability in FTP server allows remote attackers to read arbitrary files via "..\" sequences in a GET request.
CVE-2002-1178	Directory traversal vulnerability in servlet allows remote attackers to execute arbitrary commands via "..\" sequences in an HTTP request.

CVE-2002-1987	Protection mechanism checks for "/.." but doesn't account for Windows-specific "\.." allowing read of arbitrary files.
CVE-2005-2142	Directory traversal vulnerability in FTP server allows remote authenticated attackers to list arbitrary directories via a "\.." sequence in an LS command.
CVE-2002-0160	The administration function in Access Control Server allows remote attackers to read HTML, Java class, and image files outside the web root via a "..\.." sequence in the URL to port 2002.
CVE-2001-0467	"\..." in web server
CVE-2001-0963	"..." in cd command in FTP server
CVE-2001-1193	"..." in cd command in FTP server
CVE-2001-1131	"..." in cd command in FTP server
CVE-2001-0480	read of arbitrary files and directories using GET or CD with "..." in Windows-based FTP server.
CVE-2002-0288	read files using "." and Unicode-encoded "/" or "\" characters in the URL.
CVE-2003-0313	Directory listing of web server using "..."
CVE-2005-1658	Triple dot
CVE-2000-0240	read files via "/........../" in URL
CVE-2000-0773	read files via "...." in web server
CVE-1999-1082	read files via "......" in web server (doubled triple dot?)
CVE-2004-2121	read files via "......" in web server (doubled triple dot?)
CVE-2001-0491	multiple attacks using "..", "...", and "...." in different commands
CVE-2001-0615	"..." or "...." in chat server
CVE-2005-2169	chain: ".../.../" bypasses protection mechanism using regexp's that remove "../" resulting in collapse into an unsafe value "../" (CWE-182) and resultant path traversal.
CVE-2005-0202	".../..../" bypasses regexp's that remove "./" and "../"
CVE-2004-1670	Mail server allows remote attackers to create arbitrary directories via a ".." or rename arbitrary files via a "..../" in user supplied parameters.

Weakness Ordinalities

Ordinality	Description
Primary	*(where the weakness exists independent of other weaknesses)*
Resultant	*(where the weakness is typically related to the presence of some other weaknesses)*

Detection Methods

Method	Details
Automated Static Analysis	Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.) **Effectiveness: High**

Functional Areas

- File Processing

Affected Resources

- File or Directory

Memberships

Nature	Type	ID	Name
MemberOf	V	884	CWE Cross-section
MemberOf	C	981	SFP Secondary Cluster: Path Traversal
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control
MemberOf	C	1404	Comprehensive Categorization: File Handling
MemberOf	C	1436	OWASP Top Ten 2025 Category A01:2025 - Broken Access Control

Vulnerability Mapping Notes

Usage	**ALLOWED** *(this CWE ID may be used to map to real-world vulnerabilities)*
Reason	Acceptable-Use
Rationale	This CWE entry is at the Base level of abstraction, which is a preferred level of abstraction for mapping to the root causes of vulnerabilities.
Comments	Carefully read both the name and description to ensure that this mapping is an appropriate fit. Do not try to 'force' a mapping to a lower-level Base/Variant simply to comply with this preferred level of abstraction.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Relative Path Traversal
Software Fault Patterns	SFP16		Path Traversal

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
CAPEC-139	Relative Path Traversal
CAPEC-76	Manipulating Web Input to File System Calls

References

[REF-192]	OWASP. *"OWASP Attack listing".* <http://www.owasp.org/index.php/Relative_Path_Traversal>.
[REF-62]	Mark Dowd, John McDonald and Justin Schuh. *"The Art of Software Security Assessment".* Chapter 9, "Filenames and Paths", Page 503. 1st Edition. Addison Wesley. 2006.
[REF-1282]	Snyk. *"Zip Slip Vulnerability".* 2018-06-05. <https://security.snyk.io/research/zip-slip-vulnerability>.
[REF-1448]	Cybersecurity and Infrastructure Security Agency. *"Secure by Design Alert: Eliminating Directory Traversal Vulnerabilities in Software".* 2024-05-02. <https://www.cisa.gov/resources-tools/resources/secure-design-alert-eliminating-directory-traversal-vulnerabilities-software>. (*URL validated: 2024-07-14*)
[REF-1481]	D3FEND. *"D3FEND: Application Layer Firewall".* <https://d3fend.mitre.org/dao/artifact/d3f:ApplicationLayerFirewall/>. (*URL validated: 2025-09-06*)

Content History

Submissions

Submission Date	Submitter	Organization
2006-07-19 *(CWE Draft 3, 2006-07-19)*	PLOVER	

Contributions

Contribution Date	Contributor	Organization
2022-07-11	Nick Johnston *Identified weakness in Perl demonstrative example*	

Modifications