CWE-497: Exposure of Sensitive System Information to an Unauthorized Control Sphere

Weakness ID: 497
Vulnerability Mapping: ALLOWED
Abstraction: Base

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

Description

The product does not properly prevent sensitive system-level information from being accessed by unauthorized actors who do not have the same level of access to the underlying system as the product does.

Extended Description

Network-based products, such as web applications, often run on top of an operating system or similar environment. When the product communicates with outside parties, details about the underlying system are expected to remain hidden, such as path names for data files, other OS users, installed packages, the application environment, etc. This system information may be provided by the product itself, or buried within diagnostic or debugging messages. Debugging information helps an adversary learn about the system and form an attack plan.

An information exposure occurs when system data or debugging information leaves the program through an output stream or logging function that makes it accessible to unauthorized parties. Using other weaknesses, an attacker could cause errors to occur; the response to these errors can reveal detailed system information, along with other impacts. An attacker can use messages that reveal technologies, operating systems, and product versions to tune the attack against known vulnerabilities in these technologies. A product may use diagnostic methods that provide significant implementation details such as stack traces as part of its error handling mechanism.

Common Consequences

Scope	Impact	Likelihood
Confidentiality	**Technical Impact:** *Read Application Data*	

Potential Mitigations

Phases: Architecture and Design; Implementation

Production applications should never use methods that generate internal details such as stack traces and error messages unless that information is directly committed to a log that is not viewable by the end user. All error message text should be HTML entity encoded before being written to the log file to protect against potential cross-site scripting attacks against the viewer of the logs

Relationships

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

Nature	Type	ID	Name
ChildOf	B	200	Exposure of Sensitive Information to an Unauthorized Actor
ParentOf	B	214	Invocation of Process Using Visible Sensitive Information
ParentOf	V	548	Exposure of Information Through Directory Listing

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name
MemberOf	C	199	Information Management Errors

Modes Of Introduction

Phase	Note
Implementation	

Applicable Platforms

Languages

Class: Not Language-Specific *(Undetermined Prevalence)*

Demonstrative Examples

Example 1

The following code prints the path environment variable to the standard error stream:

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

```
char* path = getenv("PATH");
...
sprintf(stderr, "cannot find exe on path %s\n", path);
```

Example 2

This code prints all of the running processes belonging to the current user.

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

```
//assume getCurrentUser() returns a username that is guaranteed to be alphanumeric (avoiding CWE-78)
$userName = getCurrentUser();
$command = 'ps aux | grep ' . $userName;
system($command);
```

If invoked by an unauthorized web user, it is providing a web page of potentially sensitive information on the underlying system, such as command-line arguments (CWE-497). This program is also potentially vulnerable to a PATH based attack (CWE-426), as an attacker may be able to create malicious versions of the ps or grep commands. While the program does not explicitly raise privileges to run the system commands, the PHP interpreter may by default be running with higher privileges than users.

Example 3

The following code prints an exception to the standard error stream:

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

```
try {
  ...
} catch (Exception e) {
  e.printStackTrace();
}
```

(bad code)

```
try {
  ...
} catch (Exception e) {
  Console.Writeline(e);
}
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system will be vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Example 4

The following code constructs a database connection string, uses it to create a new connection to the database, and prints it to the console.

Example Language: **C#** *(bad code)*

```
string cs="database=northwind; server=mySQLServer...";
SqlConnection conn=new SqlConnection(cs);
...
Console.Writeline(cs);
```

Depending on the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Observed Examples

Reference	Description
CVE-2021-32638	Code analysis product passes access tokens as a command-line parameter or through an environment variable, making them visible to other processes via the ps command.

Detection Methods

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

Memberships

Nature	Type	ID	Name
MemberOf	C	485	7PK - Encapsulation
MemberOf	C	851	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 8 - Exceptional Behavior (ERR)
MemberOf	C	880	CERT C++ Secure Coding Section 12 - Exceptions and Error Handling (ERR)
MemberOf	C	963	SFP Secondary Cluster: Exposed Data
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure

Vulnerability Mapping Notes

Usage: ALLOWED *(this CWE ID could 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
7 Pernicious Kingdoms			System Information Leak
The CERT Oracle Secure Coding Standard for Java (2011)	ERR01-J		Do not allow exceptions to expose sensitive information
Software Fault Patterns	SFP23		Exposed Data

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
CAPEC-170	Web Application Fingerprinting
CAPEC-694	System Location Discovery

References

[REF-6] Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. NIST. 2005-11-07. <https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf>.

Content History

Submissions

Submission Date	Submitter	Organization
2006-07-19 *(CWE Draft 3, 2006-07-19)*	7 Pernicious Kingdoms	

Modifications

Previous Entry Names

Page Last Updated: July 16, 2024