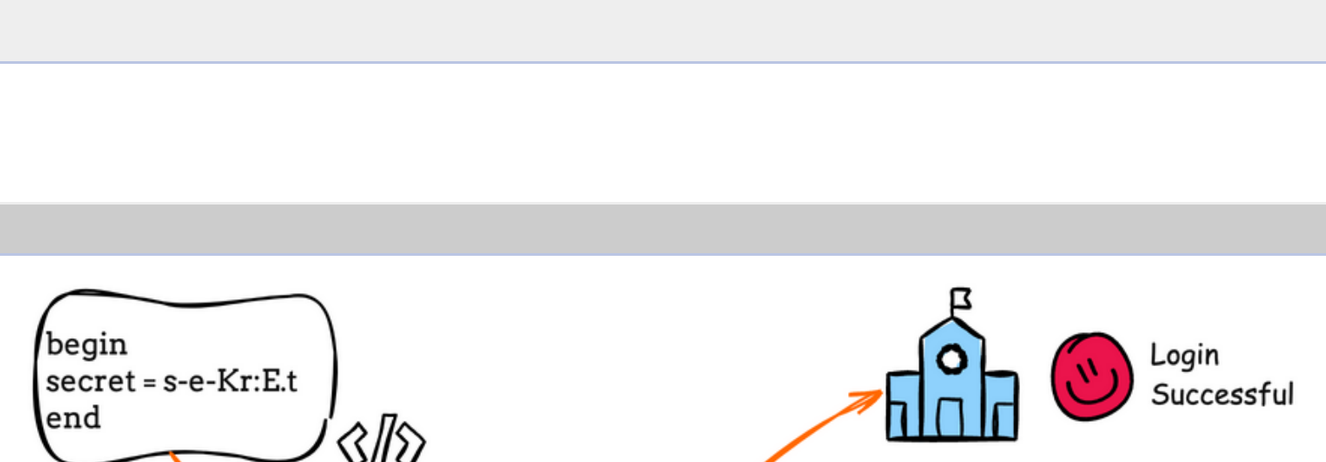
CWE-798: Use of Hard-coded Credentials

Weakness ID: 798
Vulnerability Mapping: ALLOWED
Abstraction: Base

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

Description

The product contains hard-coded credentials, such as a password or cryptographic key.



Extended Description

There are two main variations:

- Inbound: the product contains an authentication mechanism that checks the input credentials against a hard-coded set of credentials. In this variant, a default administration account is created, and a simple password is hard-coded into the product and associated with that account. This hard-coded password is the same for each installation of the product, and it usually cannot be changed or disabled by system administrators without manually modifying the program, or otherwise patching the product. It can also be difficult for the administrator to detect.
- Outbound: the product connects to another system or component, and it contains hard-coded credentials for connecting to that component. This variant applies to front-end systems that authenticate with a back-end service. The back-end service may require a fixed password that can be easily discovered. The programmer may simply hard-code those back-end credentials into the front-end product.

Common Consequences

Scope	Impact	Likelihood
Access Control	**Technical Impact:** *Bypass Protection Mechanism* If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question. Any user of the product that hard-codes passwords may be able to extract the password. Client-side systems with hard-coded passwords pose even more of a threat, since the extraction of a password from a binary is usually very simple.	
Integrity Confidentiality Availability Access Control Other	**Technical Impact:** *Read Application Data; Gain Privileges or Assume Identity; Execute Unauthorized Code or Commands; Other* This weakness can lead to the exposure of resources or functionality to unintended actors, possibly providing attackers with sensitive information or even execute arbitrary code. If the password is ever discovered or published (a common occurrence on the Internet), then anybody with knowledge of this password can access the product. Finally, since all installations of the product will have the same password, even across different organizations, this enables massive attacks such as worms to take place.	

Potential Mitigations

Phase: Architecture and Design

For outbound authentication: store passwords, keys, and other credentials outside of the code in a strongly-protected, encrypted configuration file or database that is protected from access by all outsiders, including other local users on the same system. Properly protect the key (CWE-320). If you cannot use encryption to protect the file, then make sure that the permissions are as restrictive as possible [REF-7].

In Windows environments, the Encrypted File System (EFS) may provide some protection.

Phase: Architecture and Design

For inbound authentication: Rather than hard-code a default username and password, key, or other authentication credentials for first time logins, utilize a "first login" mode that requires the user to enter a unique strong password or key.

Phase: Architecture and Design

If the product must contain hard-coded credentials or they cannot be removed, perform access control checks and limit which entities can access the feature that requires the hard-coded credentials. For example, a feature might only be enabled through the system console instead of through a network connection.

Phase: Architecture and Design

For inbound authentication using passwords: apply strong one-way hashes to passwords and store those hashes in a configuration file or database with appropriate access control. That way, theft of the file/database still requires the attacker to try to crack the password. When handling an incoming password during authentication, take the hash of the password and compare it to the saved hash.

Use randomly assigned salts for each separate hash that is generated. This increases the amount of computation that an attacker needs to conduct a brute-force attack, possibly limiting the effectiveness of the rainbow table method.

Phase: Architecture and Design

For front-end to back-end connections: Three solutions are possible, although none are complete.

- The first suggestion involves the use of generated passwords or keys that are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals.
- Next, the passwords or keys should be limited at the back end to only performing actions valid for the front end, as opposed to having full access.
- Finally, the messages sent should be tagged and checksummed with time sensitive values so as to prevent replay-style attacks.

Relationships

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

Nature	Type	ID	Name
ChildOf	B	344	Use of Invariant Value in Dynamically Changing Context
ChildOf	C	671	Lack of Administrator Control over Security
ChildOf	C	1391	Use of Weak Credentials
ParentOf	V	259	Use of Hard-coded Password
ParentOf	V	321	Use of Hard-coded Cryptographic Key
PeerOf	B	257	Storing Passwords in a Recoverable Format

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

Nature	Type	ID	Name
MemberOf	C	255	Credentials Management Errors
MemberOf	C	320	Key Management Errors

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)
Relevant to the view "Architectural Concepts" (CWE-1008)
Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)
Relevant to the view "CISQ Data Protection Measures" (CWE-1340)

Modes Of Introduction

Phase	Note
Architecture and Design	REALIZATION: This weakness is caused during implementation of an architectural security tactic.

Applicable Platforms

Languages

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

Technologies

Class: Mobile *(Undetermined Prevalence)*

Class: ICS/OT *(Often Prevalent)*

Likelihood Of Exploit

High

Demonstrative Examples

Example 1

The following code uses a hard-coded password to connect to a database:

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

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```

This is an example of an external hard-coded password on the client-side of a connection. This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is no going back from the database user "scott" with a password of "tiger" unless the program is patched. A devious employee with access to this information can use it to break into the system. Even worse, if attackers have access to the bytecode for application, they can use the javap -c command to access the disassembled code, which will contain the values of the passwords used. The result of this operation might look something like the following for the example above:

(attack code)

```
javap -c ConnMngr.class
  22: ldc #36; //String jdbc:mysql://ixne.com/rxsql
  24: ldc #38; //String scott
  26: ldc #17; //String tiger
```

Example 2

The following code is an example of an internal hard-coded password in the back-end:

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

```
int VerifyAdmin(char *password) {
  if (strcmp(password, "Mew!")) {
    printf("Incorrect Password!\n");
    return(0)
  }
  printf("Entering Diagnostic Mode...\n");
  return(1);
}
```

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

```
int VerifyAdmin(String password) {
  if (!password.equals("Mew!")) {
    return(0)
  }
  //Diagnostic Mode
  return(1);
}
```

Every instance of this program can be placed into diagnostic mode with the same password. Even worse is the fact that if this program is distributed as a binary-only distribution, it is very difficult to change that password or disable this "functionality."

Example 3

The following code examples attempt to verify a password using a hard-coded cryptographic key.

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

```
int VerifyAdmin(char *password) {
  if (strcmp(password,"68af404b513073584c4b6f22b6c63e6b")) {

    printf("Incorrect Password!\n");
    return(0);
  }
  printf("Entering Diagnostic Mode...\n");
  return(1);
}
```

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

```
public boolean VerifyAdmin(String password) {
  if (password.equals("68af404b513073584c4b6f22b6c63e6b")) {
    System.out.println("Entering Diagnostic Mode...");
    return true;
  }
  System.out.println("Incorrect Password!");
  return false;
```

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

```
int VerifyAdmin(String password) {
  if (password.Equals("68af404b513073584c4b6f22b6c63e6b")) {
    Console.WriteLine("Entering Diagnostic Mode...");
    return(1);
  }
  Console.WriteLine("Incorrect Password!");
  return(0);
}
```

The cryptographic key is within a hard-coded string value that is compared to the password. It is likely that an attacker will be able to read the key and compromise the system.

Example 4

The following examples show a portion of properties and configuration files for Java and ASP.NET applications. The files include username and password information but they are stored in cleartext.

This Java example shows a properties file with a cleartext username / password pair.

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

```
# Java Web App ResourceBundle properties file
...
webapp.ldap.username=secretUsername
webapp.ldap.password=secretPassword
...
```

The following example shows a portion of a configuration file for an ASP.Net application. This configuration file includes username and password information for a connection to a database but the pair is stored in cleartext.

Example Language: **ASP.NET** *(bad code)*

```
...
<connectionStrings>
  <add name="ud_DEV" connectionString="connectDB=uDB; uid=db2admin; pwd=password; dbalias=uDB;" providerName="System.Data.Odbc" />
</connectionStrings>
...
```

Username and password information should not be included in a configuration file or a properties file in cleartext as this will allow anyone who can read the file access to the resource. If possible, encrypt this information.

Example 5

In 2022, the OT:ICEFALL study examined products by 10 different Operational Technology (OT) vendors. The researchers reported 56 vulnerabilities and said that the products were "insecure by design" [REF-1283]. If exploited, these vulnerabilities often allowed adversaries to change how the products operated, ranging from denial of service to changing the code that the products executed. Since these products were often used in industries such as power, electrical, water, and others, there could even be safety implications.

Multiple vendors used hard-coded credentials in their OT products.

Observed Examples

Reference	Description
CVE-2022-29953	Condition Monitor firmware has a maintenance interface with hard-coded credentials
CVE-2022-29960	Engineering Workstation uses hard-coded cryptographic keys that could allow for unathorized filesystem access and privilege escalation
CVE-2022-29964	Distributed Control System (DCS) has hard-coded passwords for local shell access
CVE-2022-30997	Programmable Logic Controller (PLC) has a maintenance service that uses undocumented, hard-coded credentials
CVE-2022-30314	Firmware for a Safety Instrumented System (SIS) has hard-coded credentials for access to boot configuration
CVE-2022-30271	Remote Terminal Unit (RTU) uses a hard-coded SSH private key that is likely to be used in typical deployments
CVE-2021-37555	Telnet service for IoT feeder for dogs and cats has hard-coded password [REF-1288]
CVE-2021-35033	Firmware for a WiFi router uses a hard-coded password for a BusyBox shell, allowing bypass of authentication through the UART port
CVE-2012-3503	Installation script has a hard-coded secret token value, allowing attackers to bypass authentication
CVE-2010-2772	SCADA system uses a hard-coded password to protect back-end database containing authorization information, exploited by Stuxnet worm
CVE-2010-2073	FTP server library uses hard-coded usernames and passwords for three default accounts
CVE-2010-1573	Chain: Router firmware uses hard-coded username and password for access to debug functionality, which can be used to execute arbitrary code
CVE-2008-2369	Server uses hard-coded authentication key
CVE-2008-0961	Backup product uses hard-coded username and password, allowing attackers to bypass authentication via the RPC interface
CVE-2008-1160	Security appliance uses hard-coded password allowing attackers to gain root access
CVE-2006-7142	Drive encryption product stores hard-coded cryptographic keys for encrypted configuration files in executable programs
CVE-2005-3716	VoIP product uses hard-coded public credentials that cannot be changed, which allows attackers to obtain sensitive information
CVE-2005-3803	VoIP product uses hard coded public and private SNMP community strings that cannot be changed, which allows remote attackers to obtain sensitive information
CVE-2005-0496	Backup product contains hard-coded credentials that effectively serve as a back door, which allows remote attackers to access the file system

Weakness Ordinalities

Ordinality	Description
Primary	*(where the weakness exists independent of other weaknesses)*

Detection Methods

Black Box

Credential storage in configuration files is findable using black box methods, but the use of hard-coded credentials for an incoming authentication routine typically involves an account that is not visible outside of the code.

Effectiveness: Moderate

Automated Static Analysis

Automated white box techniques have been published for detecting hard-coded credentials for incoming authentication, but there is some expert disagreement regarding their effectiveness and applicability to a broad range of methods.

Manual Static Analysis

This weakness may be detectable using manual code analysis. Unless authentication is decentralized and applied throughout the product, there can be sufficient time for the analyst to find incoming authentication routines and examine the program logic looking for usage of hard-coded credentials. Configuration files could also be analyzed.

Note: These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Manual Dynamic Analysis

For hard-coded credentials in incoming authentication: use monitoring tools that examine the product's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the product was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.

Attach the monitor to the process and perform a login. Using call trees or similar artifacts from the output, examine the associated behaviors and see if any of them appear to be comparing the input to a fixed string or value.

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful:

Cost effective for partial coverage:

- Bytecode Weakness Analysis - including disassembler + source code weakness analysis
- Binary Weakness Analysis - including disassembler + source code weakness analysis

Effectiveness: SOAR Partial

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful:

Highly cost effective:

- Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies

Effectiveness: High

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful:

Cost effective for partial coverage:

- Network Sniffer
- Forced Path Execution

Effectiveness: SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful:

Highly cost effective:

- Focused Manual Spotcheck - Focused manual analysis of source
- Manual Source Code Review (not inspections)

Effectiveness: High

Automated Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful:

Highly cost effective:

- Source code Weakness Analyzer
- Context-configured Source Code Weakness Analyzer

Effectiveness: High

Automated Static Analysis

According to SOAR, the following detection techniques may be useful:

Cost effective for partial coverage:

- Configuration Checker

Effectiveness: SOAR Partial

Architecture or Design Review

According to SOAR, the following detection techniques may be useful:

Highly cost effective:

- Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)
- Formal Methods / Correct-By-Construction

Effectiveness: High

Memberships

Nature	Type	ID	Name
MemberOf	C	254	7PK - Security Features
MemberOf	C	724	OWASP Top Ten 2004 Category A3 - Broken Authentication and Session Management
MemberOf	C	753	2009 Top 25 - Porous Defenses
MemberOf	C	803	2010 Top 25 - Porous Defenses
MemberOf	C	812	OWASP Top Ten 2010 Category A3 - Broken Authentication and Session Management
MemberOf	C	861	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 18 - Miscellaneous (MSC)
MemberOf	C	866	2011 Top 25 - Porous Defenses
MemberOf	V	884	CWE Cross-section
MemberOf	C	1131	CISQ Quality Measures (2016) - Security
MemberOf	C	1152	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 49. Miscellaneous (MSC)
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors
MemberOf	C	1308	CISQ Quality Measures - Security
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	V	1340	CISQ Data Protection Measures
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	C	1353	OWASP Top Ten 2021 Category A07:2021 - Identification and Authentication Failures
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	C	1396	Comprehensive Categorization: Access Control
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses

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.

Notes

Maintenance

The Taxonomy_Mappings to ISA/IEC 62443 were added in CWE 4.10, but they are still under review and might change in future CWE versions. These draft mappings were performed by members of the "Mapping CWE to 62443" subgroup of the CWE-CAPEC ICS/OT Special Interest Group (SIG), and their work is incomplete as of CWE 4.10. The mappings are included to facilitate discussion and review by the broader ICS/OT community, and they are likely to change in future CWE versions.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MSC03-J		Never hard code sensitive information
OMG ASCSM	ASCSM-CWE-798		
ISA/IEC 62443	Part 3-3		Req SR 1.5
ISA/IEC 62443	Part 4-2		Req CR 1.5

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
CAPEC-191	Read Sensitive Constants Within an Executable
CAPEC-70	Try Common or Default Usernames and Passwords

References

[REF-7] Michael Howard and David LeBlanc. "Writing Secure Code". Chapter 8, "Key Management Issues" Page 272. 2nd Edition. Microsoft Press. 2002-12-04. <https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223>.

[REF-729] Johannes Ullrich. "Top 25 Series - Rank 11 - Hardcoded Credentials". SANS Software Security Institute. 2010-03-10. <https://www.sans.org/blog/top-25-series-rank-11-hardcoded-credentials/>. *URL validated: 2023-04-07.*

[REF-172] Chris Wysopal. "Mobile App Top 10 List". 2010-12-13. <https://www.veracode.com/blog/2010/12/mobile-app-top-10-list>. *URL validated: 2023-04-07.*

[REF-962] Object Management Group (OMG). "Automated Source Code Security Measure (ASCSM)". ASCSM-CWE-798. 2016-01. <http://www.omg.org/spec/ASCSM/1.0/>.

[REF-1283] Forescout Vedere Labs. "OT:ICEFALL: The legacy of "insecure by design" and its implications for certifications and risk management". 2022-06-20. <https://www.forescout.com/resources/ot-icefall-report/>.

[REF-1288] Julia Lokrantz. "Ethical hacking of a Smart Automatic Feed Dispenser". 2021-06-07. <http://kth.diva-portal.org/smash/get/diva2:1561552/FULLTEXT01.pdf>.

[REF-1304] ICS-CERT. "ICS Alert (ICS-ALERT-13-164-01): Medical Devices Hard-Coded Passwords". 2013-06-13. <https://www.cisa.gov/news-events/ics-alerts/ics-alert-13-164-01>. *URL validated: 2023-04-07.*

Content History

Submissions

Submission Date	Submitter	Organization
2010-01-15 *(CWE 1.8, 2010-02-16)*	CWE Content Team More abstract entry for hard-coded password and hard-coded cryptographic key.	MITRE

Contributions

Contribution Date	Contributor	Organization
2023-01-24 *(CWE 4.10, 2023-01-31)*	"Mapping CWE to 62443" Sub-Working Group Suggested mappings to ISA/IEC 62443.	CWE-CAPEC ICS/OT SIG
2024-02-29 *(CWE 4.15, 2024-07-16)*	Abhi Balakrishnan Provided diagram to improve CWE usability	

Modifications

Page Last Updated: July 16, 2024