```
    private String name;
    private String email;
    ...
    public RegistrationForm() {
      super();
    }
    public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {...}
    // getter and setter methods for private variables
    ...
}
```

Note that the ValidatorForm class itself extends the ActionForm class within the Struts framework API.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Form Bean Does Not Extend Validation Class
Software Fault Patterns	SFP24		Tainted input to command

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-105: Struts: Form Field Without Validator

Weakness ID : 105
Structure : Simple
Abstraction : Variant

Description

The product has a form field that is not validated by a corresponding validation form, which can introduce other weaknesses related to insufficient input validation.

Extended Description

Omitting validation for even a single input field may give attackers the leeway they need to compromise the product. Although J2EE applications are not generally susceptible to memory corruption attacks, if a J2EE application interfaces with native code that does not perform array bounds checking, an attacker may be able to use an input validation mistake in the J2EE application to launch a buffer overflow attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	1173	Improper Use of Validation Framework	1969

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	
Integrity	Bypass Protection Mechanism *If unused fields are not validated, shared business logic in an action may allow attackers to bypass the validation checks that are performed for other uses of the form.*	

Potential Mitigations

Phase: Implementation

Validate all form fields. If a field is unused, it is still important to constrain it so that it is empty or undefined.

Demonstrative Examples

Example 1:

In the following example the Java class RegistrationForm is a Struts framework ActionForm Bean that will maintain user input data from a registration webpage for an online business site. The user will enter registration data and, through the Struts framework, the RegistrationForm bean will maintain the user data in the form fields using the private member variables. The RegistrationForm class uses the Struts validation capability by extending the ValidatorForm class and including the validation for the form fields within the validator XML file, validator.xml.

Example Language: *(Result)*

```
public class RegistrationForm extends org.apache.struts.validator.ValidatorForm {
  // private variables for registration form
  private String name;
  private String address;
  private String city;
  private String state;
  private String zipcode;
  private String phone;
  private String email;
  public RegistrationForm() {
    super();
  }
  // getter and setter methods for private variables
  ...
}
```

The validator XML file, validator.xml, provides the validation for the form fields of the RegistrationForm.

Example Language: XML *(Bad)*

```
<form-validation>
  <formset>
    <form name="RegistrationForm">
      <field property="name" depends="required">
        <arg position="0" key="prompt.name"/>
      </field>
      <field property="address" depends="required">
        <arg position="0" key="prompt.address"/>
      </field>
      <field property="city" depends="required">
        <arg position="0" key="prompt.city"/>
      </field>
      <field property="state" depends="required,mask">
        <arg position="0" key="prompt.state"/>
        <var>
          <var-name>mask</var-name>
          <var-value>[a-zA-Z]{2}</var-value>
        </var>
      </field>
      <field property="zipcode" depends="required,mask">
        <arg position="0" key="prompt.zipcode"/>
        <var>
          <var-name>mask</var-name>
          <var-value>\d{5}</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

However, in the previous example the validator XML file, validator.xml, does not provide validators for all of the form fields in the RegistrationForm. Validator forms are only provided for the first five of the seven form fields. The validator XML file should contain validator forms for all of the form fields for a Struts ActionForm bean. The following validator.xml file for the RegistrationForm class contains validator forms for all of the form fields.

Example Language: XML *(Good)*

```
<form-validation>
  <formset>
    <form name="RegistrationForm">
      <field property="name" depends="required">
        <arg position="0" key="prompt.name"/>
      </field>
      <field property="address" depends="required">
        <arg position="0" key="prompt.address"/>
      </field>
      <field property="city" depends="required">
        <arg position="0" key="prompt.city"/>
      </field>
      <field property="state" depends="required,mask">
        <arg position="0" key="prompt.state"/>
        <var>
          <var-name>mask</var-name>
          <var-value>[a-zA-Z]{2}</var-value>
        </var>
      </field>
      <field property="zipcode" depends="required,mask">
        <arg position="0" key="prompt.zipcode"/>
        <var>
          <var-name>mask</var-name>
          <var-value>\d{5}</var-value>
```

```
                </var>
            </field>
            <field property="phone" depends="required,mask">
                <arg position="0" key="prompt.phone"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>^([0-9]{3})(-)([0-9]{4}|[0-9]{4})$</var-value>
                </var>
            </field>
            <field property="email" depends="required,email">
                <arg position="0" key="prompt.email"/>
            </field>
        </form>
    </formset>
</form-validation>
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Form Field Without Validator
Software Fault Patterns	SFP24		Tainted input to command

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-106: Struts: Plug-in Framework not in Use

Weakness ID : 106
Structure : Simple
Abstraction : Variant

Description

When an application does not use an input validation framework such as the Struts Validator, there is a greater risk of introducing weaknesses related to insufficient input validation.

Extended Description

Unchecked input is the leading cause of vulnerabilities in J2EE applications. Unchecked input leads to cross-site scripting, process control, and SQL injection vulnerabilities, among others.

Although J2EE applications are not generally susceptible to memory corruption attacks, if a J2EE application interfaces with native code that does not perform array bounds checking, an attacker may be able to use an input validation mistake in the J2EE application to launch a buffer overflow attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	1173	Improper Use of Validation Framework	1969

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Use an input validation framework such as Struts.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Use an input validation framework such as Struts.

Phase: Implementation

Strategy = Input Validation

Use the Struts Validator to validate all program input before it is processed by the application. Ensure that there are no holes in the configuration of the Struts Validator. Example uses of the validator include checking to ensure that: Phone number fields contain only valid characters in phone numbers Boolean values are only "T" or "F" Free-form strings are of a reasonable length and composition

Phase: Implementation

Strategy = Libraries or Frameworks

Use the Struts Validator to validate all program input before it is processed by the application. Ensure that there are no holes in the configuration of the Struts Validator. Example uses of the validator include checking to ensure that: Phone number fields contain only valid characters in phone numbers Boolean values are only "T" or "F" Free-form strings are of a reasonable length and composition

Demonstrative Examples

Example 1:

In the following Java example the class RegistrationForm is a Struts framework ActionForm Bean that will maintain user input data from a registration webpage for an online business site. The user will enter registration data and, through the Struts framework, the RegistrationForm bean will maintain the user data.

Example Language: Java *(Bad)*

```
public class RegistrationForm extends org.apache.struts.action.ActionForm {
   // private variables for registration form
   private String name;
   private String email;
   ...
   public RegistrationForm() {
      super();
   }
   // getter and setter methods for private variables
   ...
}
```

However, the RegistrationForm class extends the Struts ActionForm class which does use the Struts validator plug-in to provide validator capabilities. In the following example, the RegistrationForm Java class extends the ValidatorForm and Struts configuration XML file, struts-config.xml, instructs the application to use the Struts validator plug-in.

Example Language: Java *(Good)*

```
public class RegistrationForm extends org.apache.struts.validator.ValidatorForm {
   // private variables for registration form
   private String name;
   private String email;
   ...
   public RegistrationForm() {
      super();
   }
   public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {...}
   // getter and setter methods for private variables
   ...
}
```

The plug-in tag of the Struts configuration XML file includes the name of the validator plug-in to be used and includes a set-property tag to instruct the application to use the file, validator-rules.xml, for default validation rules and the file, validation.XML, for custom validation.

Example Language: XML *(Good)*

```
<struts-config>
   <form-beans>
      <form-bean name="RegistrationForm" type="RegistrationForm"/>
   </form-beans>
   ...
   <!-- ======================== Validator plugin ================================= -->
   <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
      <set-property
         property="pathnames"
         value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
   </plug-in>
</struts-config>
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Plug-in Framework Not In Use

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-107: Struts: Unused Validation Form

Weakness ID : 107
Structure : Simple
Abstraction : Variant

Description

An unused validation form indicates that validation logic is not up-to-date.

Extended Description

It is easy for developers to forget to update validation logic when they remove or rename action form mappings. One indication that validation logic is not being properly maintained is the presence of an unused validation form.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	1164	Irrelevant Code	1967

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Resultant :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Potential Mitigations

Phase: Implementation

Remove the unused Validation Form from the validation.xml file.

Demonstrative Examples

Example 1:

In the following example the class RegistrationForm is a Struts framework ActionForm Bean that will maintain user input data from a registration webpage for an online business site. The user will enter registration data and, through the Struts framework, the RegistrationForm bean will maintain the user data in the form fields using the private member variables. The RegistrationForm class uses the Struts validation capability by extending the ValidatorForm class and including the validation for the form fields within the validator XML file, validator.xml.

Example Language: Java *(Bad)*

```
public class RegistrationForm extends org.apache.struts.validator.ValidatorForm {
  // private variables for registration form
  private String name;
  private String address;
  private String city;
  private String state;
  private String zipcode;
  // no longer using the phone form field
  // private String phone;
  private String email;
  public RegistrationForm() {
    super();
  }
  // getter and setter methods for private variables
  ...
}
```

Example Language: XML *(Bad)*

```
<form-validation>
  <formset>
    <form name="RegistrationForm">
      <field property="name" depends="required">
        <arg position="0" key="prompt.name"/>
      </field>
      <field property="address" depends="required">
        <arg position="0" key="prompt.address"/>
      </field>
      <field property="city" depends="required">
        <arg position="0" key="prompt.city"/>
      </field>
      <field property="state" depends="required,mask">
        <arg position="0" key="prompt.state"/>
        <var>
          <var-name>mask</var-name>
          <var-value>[a-zA-Z]{2}</var-value>
        </var>
      </field>
      <field property="zipcode" depends="required,mask">
        <arg position="0" key="prompt.zipcode"/>
        <var>
          <var-name>mask</var-name>
          <var-value>\d{5}</var-value>
        </var>
      </field>
      <field property="phone" depends="required,mask">
        <arg position="0" key="prompt.phone"/>
        <var>
          <var-name>mask</var-name>
          <var-value>^([0-9]{3})(-)([0-9]{4}|[0-9]{4})$</var-value>
        </var>
      </field>
      <field property="email" depends="required,email">
        <arg position="0" key="prompt.email"/>
      </field>
    </form>
  </formset>
</form-validation>
```

However, the validator XML file, validator.xml, for the RegistrationForm class includes the validation form for the user input form field "phone" that is no longer used by the input form and the RegistrationForm class. Any validation forms that are no longer required should be removed from the validator XML file, validator.xml.

The existence of unused forms may be an indication to attackers that this code is out of date or poorly maintained.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Unused Validation Form

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-108: Struts: Unvalidated Action Form

Weakness ID : 108
Structure : Simple
Abstraction : Variant

Description

Every Action Form must have a corresponding validation form.

Extended Description

If a Struts Action Form Mapping specifies a form, it must have a validation form defined under the Struts Validator.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	1173	Improper Use of Validation Framework	1969

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Other *If an action form mapping does not have a validation form defined, it may be vulnerable to a number of attacks that rely on unchecked input. Unchecked input is the root cause of some of today's worst and most common software security problems. Cross-site scripting, SQL injection, and process control vulnerabilities all stem from incomplete or absent input validation.*	
Confidentiality Integrity Availability Other	Other *Although J2EE applications are not generally susceptible to memory corruption attacks, if a J2EE application interfaces with native code that does not perform array bounds checking, an attacker may be able to use an input validation mistake in the J2EE application to launch a buffer overflow attack.*	

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Map every Action Form to a corresponding validation form. An action or a form may perform validation in other ways, but the Struts Validator provides an excellent way to verify that all input receives at least a basic level of validation. Without this approach, it is difficult, and often impossible, to establish with a high level of confidence that all input is validated.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Unvalidated Action Form
Software Fault Patterns	SFP24		Tainted input to command

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-109: Struts: Validator Turned Off

Weakness ID : 109
Structure : Simple
Abstraction : Variant

Description

Automatic filtering via a Struts bean has been turned off, which disables the Struts Validator and custom validation logic. This exposes the application to other weaknesses related to insufficient input validation.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	1173	Improper Use of Validation Framework	1969

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Implementation

Ensure that an action form mapping enables validation. Set the validate field to true.

Demonstrative Examples

Example 1:

This mapping defines an action for a download form:

Example Language: XML *(Bad)*

```
<action path="/download"
type="com.website.d2.action.DownloadAction"
name="downloadForm"
scope="request"
input=".download"
```

```
validate="false">
</action>
```

This mapping has disabled validation. Disabling validation exposes this action to numerous types of attacks.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Notes

Other

The Action Form mapping in the demonstrative example disables the form's validate() method. The Struts bean: write tag automatically encodes special HTML characters, replacing a < with "&lt;" and a > with "&gt;". This action can be disabled by specifying filter="false" as an attribute of the tag to disable specified JSP pages. However, being disabled makes these pages susceptible to cross-site scripting attacks. An attacker may be able to insert malicious scripts as user input to write to these JSP pages.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Validator Turned Off
Software Fault Patterns	SFP24		Tainted input to command

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-110: Struts: Validator Without Form Field

Weakness ID : 110
Structure : Simple
Abstraction : Variant

Description

Validation fields that do not appear in forms they are associated with indicate that the validation logic is out of date.

Extended Description

It is easy for developers to forget to update validation logic when they make changes to an ActionForm class. One indication that validation logic is not being properly maintained is inconsistencies between the action form and the validation form.

Although J2EE applications are not generally susceptible to memory corruption attacks, if a J2EE application interfaces with native code that does not perform array bounds checking, an attacker may be able to use an input validation mistake in the J2EE application to launch a buffer overflow attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	1164	Irrelevant Code	1967

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	●	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Other *It is critically important that validation logic be maintained and kept in sync with the rest of the application. Unchecked input is the root cause of some of today's worst and most common software security problems. Cross-site scripting, SQL injection, and process control vulnerabilities all stem from incomplete or absent input validation.*	

Detection Methods

Automated Static Analysis

To find the issue in the implementation, manual checks or automated static analysis could be applied to the XML configuration files.

Effectiveness = Moderate

Manual Static Analysis

To find the issue in the implementation, manual checks or automated static analysis could be applied to the XML configuration files.

Effectiveness = Moderate

Demonstrative Examples

Example 1:

This example shows an inconsistency between an action form and a validation form. with a third field.

This first block of code shows an action form that has two fields, startDate and endDate.

Example Language: Java *(Bad)*

```
public class DateRangeForm extends ValidatorForm {
  String startDate, endDate;
```

```
    public void setStartDate(String startDate) {
        this.startDate = startDate;
    }
    public void setEndDate(String endDate) {
        this.endDate = endDate;
    }
}
```

This second block of related code shows a validation form with a third field: scale. The presence of the third field suggests that DateRangeForm was modified without taking validation into account.

Example Language: XML *(Bad)*

```
<form name="DateRangeForm">
    <field property="startDate" depends="date">
        <arg0 key="start.date"/>
    </field>
    <field property="endDate" depends="date">
        <arg0 key="end.date"/>
    </field>
    <field property="scale" depends="integer">
        <arg0 key="range.scale"/>
    </field>
</form>
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Struts: Validator Without Form Field
Software Fault Patterns	SFP24		Tainted input to command

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-111: Direct Use of Unsafe JNI

Weakness ID : 111
Structure : Simple
Abstraction : Variant

Description

When a Java application uses the Java Native Interface (JNI) to call code written in another programming language, it can expose the application to weaknesses in that code, even if those weaknesses cannot occur in Java.

Extended Description

Many safety features that programmers may take for granted do not apply for native code, so you must carefully review all such code for potential problems. The languages used to implement native code may be more susceptible to buffer overflows and other attacks. Native code is unprotected by the security features enforced by the runtime environment, such as strong typing and array bounds checking.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	695	Use of Low-Level Functionality	1524

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

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

Potential Mitigations

Phase: Implementation

Implement error handling around the JNI call.

Phase: Implementation

Strategy = Refactoring

Do not use JNI calls if you don't trust the native library.

Phase: Implementation

Strategy = Refactoring

Be reluctant to use JNI calls. A Java API equivalent may exist.

Demonstrative Examples

Example 1:

The following code defines a class named Echo. The class declares one native method (defined below), which uses C to echo commands entered on the console back to the user. The following C code defines the native method implemented in the Echo class:

Example Language: Java *(Bad)*

```
class Echo {
  public native void runEcho();
  static {
    System.loadLibrary("echo");
  }
  public static void main(String[] args) {
    new Echo().runEcho();
  }
}
```

Example Language: C *(Bad)*

```
#include <jni.h>
#include "Echo.h"//the java class above compiled with javah
#include <stdio.h>
JNIEXPORT void JNICALL
Java_Echo_runEcho(JNIEnv *env, jobject obj)
{
  char buf[64];
  gets(buf);
  printf(buf);
}
```

Because the example is implemented in Java, it may appear that it is immune to memory issues like buffer overflow vulnerabilities. Although Java does do a good job of making memory operations safe, this protection does not extend to vulnerabilities occurring in source code written in other languages that are accessed using the Java Native Interface. Despite the memory protections offered in Java, the C code in this example is vulnerable to a buffer overflow because it makes use of gets(), which does not check the length of its input.

The Sun Java(TM) Tutorial provides the following description of JNI [See Reference]: The JNI framework lets your native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

The vulnerability in the example above could easily be detected through a source code audit of the native method implementation. This may not be practical or possible depending on the availability of the C source code and the way the project is built, but in many cases it may suffice. However, the ability to share objects between Java and native methods expands the potential risk to much more insidious cases where improper data handling in Java may lead to unexpected vulnerabilities in native code or unsafe operations in native code corrupt data structures in Java. Vulnerabilities in native code accessed through a Java application are typically exploited in the same manner as they are in applications written in the native language. The only challenge to such an attack is for the attacker to identify that the Java application uses native code to perform certain operations. This can be accomplished in a variety of ways, including identifying specific behaviors that are often implemented with native code or by exploiting a system information exposure in the Java application that reveals its use of JNI [See Reference].

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	859	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 16 - Platform Security (SEC)	844	2369
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1151	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 17. Java Native Interface (JNI)	1133	2453
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Unsafe JNI
The CERT Oracle Secure Coding Standard for Java (2011)	SEC08-J		Define wrappers around native methods
SEI CERT Oracle Coding Standard for Java	JNI01-J		Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance (loadLibrary)
SEI CERT Oracle Coding Standard for Java	JNI00-J	Imprecise	Define wrappers around native methods
Software Fault Patterns	SFP3		Use of an improper API

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

[REF-41]Fortify Software. "Fortify Descriptions". < http://vulncat.fortifysoftware.com >.

[REF-42]Beth Stearns. "The Java(TM) Tutorial: The Java Native Interface". 2005. Sun Microsystems. < http://www.eg.bucknell.edu/~mead/Java-tutorial/native1.1/index.html >.

CWE-112: Missing XML Validation

Weakness ID : 112
Structure : Simple
Abstraction : Base

Description

The product accepts XML from an untrusted source but does not validate the XML against the proper schema.

Extended Description

Most successful attacks begin with a violation of the programmer's assumptions. By accepting an XML document without validating it against a DTD or XML schema, the programmer leaves a door open for attackers to provide unexpected, unreasonable, or malicious input.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	1286	Improper Validation of Syntactic Correctness of Input	2136

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

Nature	Type	ID	Name	Page
MemberOf	C	1215	Data Validation Issues	2478

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

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

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Always validate XML input against a known XML Schema or DTD. It is not possible for an XML parser to validate all aspects of a document's content because a parser cannot understand the complete semantics of the data. However, a parser can do a complete and thorough job of checking the document's structure and therefore guarantee to the code that processes the document that the content is well-formed.

Demonstrative Examples

Example 1:

The following code loads and parses an XML file.

Example Language: Java *(Bad)*

```
// Read DOM
try {
  ...
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setValidating( false );
```

```
    ....
    c_dom = factory.newDocumentBuilder().parse( xmlFile );
} catch(Exception ex) {
    ...
}
```

The XML file is loaded without validating it against a known XML Schema or DTD.

Example 2:

The following code creates a DocumentBuilder object to be used in building an XML document.

Example Language: Java *(Bad)*

```
DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();
builderFactory.setNamespaceAware(true);
DocumentBuilder builder = builderFactory.newDocumentBuilder();
```

The DocumentBuilder object does not validate an XML document against a schema, making it possible to create an invalid XML document.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Missing XML Validation
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
230	Serialized Data with Nested Payloads
231	Oversized Serialized Data Payloads

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Request/Response Splitting')

Weakness ID : 113
Structure : Simple
Abstraction : Variant

Description

The product receives data from an HTTP agent/component (e.g., web server, proxy, browser, etc.), but it does not neutralize or incorrectly neutralizes CR and LF characters before the data is included in outgoing HTTP headers.

Extended Description

HTTP agents or components may include a web server, load balancer, reverse proxy, web caching proxy, application firewall, web browser, etc. Regardless of the role, they are expected to maintain coherent, consistent HTTP communication state across all components. However, including unexpected data in an HTTP header allows an attacker to specify the entirety of the HTTP message that is rendered by the client HTTP agent (e.g., web browser) or back-end HTTP agent (e.g., web server), whether the message is part of a request or a response.

When an HTTP request contains unexpected CR and LF characters, the server may respond with an output stream that is interpreted as "splitting" the stream into two different HTTP messages instead of one. CR is carriage return, also given by %0d or \r, and LF is line feed, also given by %0a or \n.

In addition to CR and LF characters, other valid/RFC compliant special characters and unique character encodings can be utilized, such as HT (horizontal tab, also given by %09 or \t) and SP (space, also given as + sign or %20).

These types of unvalidated and unexpected data in HTTP message headers allow an attacker to control the second "split" message to mount attacks such as server-side request forgery, cross-site scripting, and cache poisoning attacks.

HTTP response splitting weaknesses may be present when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.
2. The data is included in an HTTP response header sent to a web user without neutralizing malicious characters that can be interpreted as separator characters for headers.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	436	Interpretation Conflict	1057
ChildOf	B	93	Improper Neutralization of CRLF Sequences ('CRLF Injection')	217
CanPrecede	B	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	163

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Applicable Platforms

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

Technology : Web Based *(Prevalence = Undetermined)*

Alternate Terms

HTTP Request Splitting :

HTTP Response Splitting :

Common Consequences

Scope	Impact	Likelihood
Integrity Access Control	Modify Application Data Gain Privileges or Assume Identity *CR and LF characters in an HTTP header may give attackers control of the remaining headers and body of the message that the application intends to send/receive, as well as allowing them to create additional messages entirely under their control.*	

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

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Construct HTTP headers very carefully, avoiding the use of non-validated input data.

Phase: 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. If an input does not strictly conform to specifications, reject it or transform it into something that conforms. 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.

Phase: Implementation

Strategy = Output Encoding

Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component.

Phase: 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.

Demonstrative Examples

Example 1:

The following code segment reads the name of the author of a weblog entry, author, from an HTTP request and sets it in a cookie header of an HTTP response.

Example Language: Java *(Bad)*

```
String author = request.getParameter(AUTHOR_PARAM);
...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

Assuming a string consisting of standard alpha-numeric characters, such as "Jane Smith", is submitted in the request the HTTP response including this cookie might take the following form:

Example Language: *(Result)*

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

However, because the value of the cookie is composed of unvalidated user input, the response will only maintain this form if the value submitted for AUTHOR_PARAM does not contain any CR and LF characters. If an attacker submits a malicious string, such as

Example Language: *(Attack)*

```
Wiley Hacker\r\nHTTP/1.1 200 OK\r\n
```

then the HTTP response would be split into two responses of the following form:

Example Language: *(Result)*

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Wiley Hacker
HTTP/1.1 200 OK
...
```

The second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability to construct arbitrary HTTP responses permits a variety of resulting attacks, including:

- cross-user defacement
- web and browser cache poisoning
- cross-site scripting
- page hijacking

Example 2:

An attacker can make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user sharing the same TCP connection with the server.

Cross-User Defacement can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server.

- In the best case, an attacker can leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application.
- In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

Example 3:

The impact of a maliciously constructed response can be magnified if it is cached, either by a web cache used by multiple users or even the browser cache of a single user.

Cache Poisoning: if a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although the user of the local browser instance will be affected.

Example 4:

Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users.

Cross-Site Scripting: cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser.

The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

Example 5:

In addition to using a vulnerable application to send malicious content to a user, the same weakness can also be leveraged to redirect sensitive content generated by the server to the attacker instead of the intended user.

Page Hijacking: by submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker can cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server to the attacker instead of the intended user.

Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

Observed Examples

Reference	Description
CVE-2020-15811	Chain: Proxy uses a substring search instead of parsing the Transfer-Encoding header (CWE-697), allowing request splitting (CWE-113) and cache poisoning *https://www.cve.org/CVERecord?id=CVE-2020-15811*
CVE-2021-41084	Scala-based HTTP interface allows request splitting and response splitting through header names, header values, status reasons, and URIs *https://www.cve.org/CVERecord?id=CVE-2021-41084*
CVE-2018-12116	Javascript-based framework allows request splitting through a path option of an HTTP request *https://www.cve.org/CVERecord?id=CVE-2018-12116*
CVE-2004-2146	Application accepts CRLF in an object ID, allowing HTTP response splitting. *https://www.cve.org/CVERecord?id=CVE-2004-2146*
CVE-2004-1656	Shopping cart allows HTTP response splitting to perform HTML injection via CRLF in a parameter for a url *https://www.cve.org/CVERecord?id=CVE-2004-1656*
CVE-2005-2060	Bulletin board allows response splitting via CRLF in parameter. *https://www.cve.org/CVERecord?id=CVE-2005-2060*
CVE-2004-2512	Response splitting via CRLF in PHPSESSID. *https://www.cve.org/CVERecord?id=CVE-2004-2512*
CVE-2005-1951	e-commerce app allows HTTP response splitting using CRLF in object id parameters *https://www.cve.org/CVERecord?id=CVE-2005-1951*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1409	Comprehensive Categorization: Injection	1400	2535

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			HTTP response splitting
7 Pernicious Kingdoms			HTTP Response Splitting
WASC	25		HTTP Response Splitting
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
31	Accessing/Intercepting/Modifying HTTP Cookies
34	HTTP Response Splitting
85	AJAX Footprinting
105	HTTP Request Splitting

References

[REF-43]OWASP. "OWASP TOP 10". 2007 May 8. < https://github.com/owasp-top/owasp-top-2007 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-1272]Robert Auger. "HTTP Request Splitting". 2011 February 1. < http://projects.webappsec.org/w/page/13246929/HTTP%20Request%20Splitting >.

CWE-114: Process Control

Weakness ID : 114
Structure : Simple
Abstraction : Class

Description

Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.

Extended Description

Process control vulnerabilities take two forms:

- An attacker can change the command that the program executes: the attacker explicitly controls what the command is.
- An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means.

Process control vulnerabilities of the first type occur when either data enters the application from an untrusted source and the data is used as part of a string representing a command that is executed by the application. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	73	External Control of File Name or Path	132

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Execute Unauthorized Code or Commands	

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

Potential Mitigations

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Libraries that are loaded should be well understood and come from a trusted source. The application can execute code contained in the native libraries, which often contain calls that are susceptible to other security problems, such as buffer overflows or command injection. All native libraries should be validated to determine if the application requires the use of the library. It is very difficult to determine what these native libraries actually do, and the potential for malicious code is high. In addition, the potential for an inadvertent mistake in these native libraries is also high, as many are written in C or C++ and may be susceptible to buffer overflow or race condition problems. To help prevent buffer overflow attacks, validate all input to native calls for content and length. If the native library does not come from a trusted source, review the source code of the library. The library should be built from the reviewed source before using it.

Demonstrative Examples

Example 1:

The following code uses System.loadLibrary() to load code from a native library named library.dll, which is normally found in a standard system directory.

Example Language: Java *(Bad)*

```
...
System.loadLibrary("library.dll");
...
```

The problem here is that System.loadLibrary() accepts a library name, not a path, for the library to be loaded. From the Java 1.4.2 API documentation this function behaves as follows [1]: A file containing native code is loaded from the local file system from a place where library files are conventionally obtained. The details of this process are implementation-dependent. The mapping from a library name to a specific filename is done in a system-specific manner. If an attacker is able to place a malicious copy of library.dll higher in the search order than file the application intends to load, then the application will load the malicious copy instead of the intended file. Because of the nature of the application, it runs with elevated privileges, which means the contents of the attacker's library.dll will now be run with elevated privileges, possibly giving them complete control of the system.

Example 2:

The following code from a privileged application uses a registry entry to determine the directory in which it is installed and loads a library file based on a relative path from the specified directory.

Example Language: C *(Bad)*

```
...
RegQueryValueEx(hkey, "APPHOME",
0, 0, (BYTE*)home, &size);
```

```
char* lib=(char*)malloc(strlen(home)+strlen(INITLIB));
if (lib) {
  strcpy(lib,home);
  strcat(lib,INITCMD);
  LoadLibrary(lib);
}
...
```

The code in this example allows an attacker to load an arbitrary library, from which code will be executed with the elevated privilege of the application, by modifying a registry key to specify a different path containing a malicious version of INITLIB. Because the program does not validate the value read from the environment, if an attacker can control the value of APPHOME, they can fool the application into running malicious code.

Example 3:

The following code is from a web-based administration utility that allows users access to an interface through which they can update their profile on the system. The utility makes use of a library named liberty.dll, which is normally found in a standard system directory.

Example Language: C *(Bad)*

```
LoadLibrary("liberty.dll");
```

The problem is that the program does not specify an absolute path for liberty.dll. If an attacker is able to place a malicious library named liberty.dll higher in the search order than file the application intends to load, then the application will load the malicious copy instead of the intended file. Because of the nature of the application, it runs with elevated privileges, which means the contents of the attacker's liberty.dll will now be run with elevated privileges, possibly giving the attacker complete control of the system. The type of attack seen in this example is made possible because of the search order used by LoadLibrary() when an absolute path is not specified. If the current directory is searched before system directories, as was the case up until the most recent versions of Windows, then this type of attack becomes trivial if the attacker can execute the program locally. The search order is operating system version dependent, and is controlled on newer operating systems by the value of the registry key: HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode

Affected Resources

- System Process

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	991	SFP Secondary Cluster: Tainted Input to Environment	888	2416
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Notes

Maintenance

CWE-114 is a Class, but it is listed a child of CWE-73 in view 1000. This suggests some abstraction problems that should be resolved in future versions.

Maintenance

This entry seems to have close relationships with CWE-426/CWE-427. It seems more attack-oriented.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Process Control

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
108	Command Line Execution through SQL Injection
640	Inclusion of Code in Existing Process

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-115: Misinterpretation of Input

Weakness ID : 115
Structure : Simple
Abstraction : Base

Description

The product misinterprets an input, whether from an attacker or another product, in a security-relevant fashion.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	436	Interpretation Conflict	1057

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

Nature	Type	ID	Name	Page
MemberOf	C	438	Behavioral Problems	2326

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Observed Examples

Reference	Description
CVE-2005-2225	Product sees dangerous file extension in free text of a group discussion, disconnects all users. *https://www.cve.org/CVERecord?id=CVE-2005-2225*
CVE-2001-0003	Product does not correctly import and process security settings from another product. *https://www.cve.org/CVERecord?id=CVE-2001-0003*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1398	Comprehensive Categorization: Component Interaction	1400	2524

Notes

Research Gap

This concept needs further study. It is likely a factor in several weaknesses, possibly resultant as well. Overlaps Multiple Interpretation Errors (MIE).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Misinterpretation Error

CWE-116: Improper Encoding or Escaping of Output

Weakness ID : 116
Structure : Simple
Abstraction : Class

Description

The product prepares a structured message for communication with another component, but encoding or escaping of the data is either missing or done incorrectly. As a result, the intended structure of the message is not preserved.

Extended Description

Improper encoding or escaping can allow attackers to change the commands that are sent to another component, inserting malicious commands instead.

Most products follow a certain protocol that uses structured messages for communication between components, such as queries or commands. These structured messages can contain raw data interspersed with metadata or control information. For example, "GET /index.html HTTP/1.1" is a structured message containing a command ("GET") with a single argument ("/index.html") and metadata about which protocol version is being used ("HTTP/1.1").

If an application uses attacker-supplied inputs to construct a structured message without properly encoding or escaping, then the attacker could insert special characters that will cause the data to be interpreted as control information or metadata. Consequently, the component that receives the output will perform the wrong operations, or otherwise interpret the data incorrectly.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓟ	707	Improper Neutralization	1546
ParentOf	Ⓑ	117	Improper Output Neutralization for Logs	288
ParentOf	Ⓥ	644	Improper Neutralization of HTTP Headers for Scripting Syntax	1422
ParentOf	Ⓑ	838	Inappropriate Encoding for Output Context	1764
CanPrecede	Ⓒ	74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	137

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ParentOf	Ⓑ	838	Inappropriate Encoding for Output Context	1764

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Often)*

Technology : Database Server *(Prevalence = Often)*

Technology : Web Server *(Prevalence = Often)*

Alternate Terms

Output Sanitization :

Output Validation :

Output Encoding :

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *The communications between components can be modified in unexpected ways. Unexpected commands can be executed, bypassing other security mechanisms. Incoming data can be misinterpreted.*	
Integrity Confidentiality Availability Access Control	Execute Unauthorized Code or Commands *The communications between components can be modified in unexpected ways. Unexpected commands can be executed, bypassing other security mechanisms. Incoming data can be misinterpreted.*	
Confidentiality	Bypass Protection Mechanism *The communications between components can be modified in unexpected ways. Unexpected commands can be executed, bypassing other security mechanisms. Incoming data can be misinterpreted.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives.

Effectiveness = Moderate

This is not a perfect solution, since 100% accuracy and coverage are not feasible.

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Potential Mitigations

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using the ESAPI Encoding control [REF-45] or a similar tool, library, or framework. These will help the programmer encode outputs in a manner less prone to error. Alternately, use built-in functions, but consider using wrappers in case those functions are discovered to have a vulnerability.

Phase: Architecture and Design

Strategy = Parameterization

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. For example, stored procedures can enforce database query structure and reduce the likelihood of SQL injection.

Phase: Architecture and Design

Phase: Implementation

Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.

Phase: Architecture and Design

In some cases, input validation may be an important strategy when output encoding is not a complete solution. For example, you may be providing the same output that will be processed by multiple consumers that use different encodings or representations. In other cases, you may be required to allow user-supplied input to contain control information, such as limited HTML tags that support formatting in a wiki or bulletin board. When this type of requirement must be met, use an extremely strict allowlist to limit which control sequences can be used. Verify that the resulting syntactic structure is what you expect. Use your normal encoding methods for the remainder of the input.

Phase: Architecture and Design

Use input validation as a defense-in-depth measure to reduce the likelihood of output encoding errors (see CWE-20).

Phase: Requirements

Fully specify which encodings are required by components that will be communicating with each other.

Phase: Implementation

When exchanging data between components, ensure that both components are using the same character encoding. Ensure that the proper encoding is applied at each interface. Explicitly set the encoding you are using whenever the protocol allows you to do so.

Demonstrative Examples

Example 1:

This code displays an email address that was submitted as part of a form.

Example Language: JSP *(Bad)*

```
<% String email = request.getParameter("email"); %>
...
Email Address: <%= email %>
```

The value read from the form parameter is reflected back to the client browser without having been encoded prior to output, allowing various XSS attacks (CWE-79).

Example 2:

Consider a chat application in which a front-end web application communicates with a back-end server. The back-end is legacy code that does not perform authentication or authorization, so the front-end must implement it. The chat protocol supports two commands, SAY and BAN, although only administrators can use the BAN command. Each argument must be separated by a single space. The raw inputs are URL-encoded. The messaging protocol allows multiple commands to be specified on the same line if they are separated by a "|" character.

First let's look at the back end command processor code

Example Language: Perl *(Bad)*

```
$inputString = readLineFromFileHandle($serverFH);
# generate an array of strings separated by the "|" character.
@commands = split(/\|/, $inputString);
foreach $cmd (@commands) {
  # separate the operator from its arguments based on a single whitespace
  ($operator, $args) = split(/ /, $cmd, 2);
  $args = UrlDecode($args);
  if ($operator eq "BAN") {
    ExecuteBan($args);
  }
  elsif ($operator eq "SAY") {
    ExecuteSay($args);
  }
}
```

The front end web application receives a command, encodes it for sending to the server, performs the authorization check, and sends the command to the server.

Example Language: Perl *(Bad)*

```
$inputString = GetUntrustedArgument("command");
($cmd, $argstr) = split(/\s+/, $inputString, 2);
# removes extra whitespace and also changes CRLF's to spaces
$argstr =~ s/\s+/ /gs;
$argstr = UrlEncode($argstr);
if (($cmd eq "BAN") && (! IsAdministrator($username))) {
  die "Error: you are not the admin.\n";
}
# communicate with file server using a file handle
```

```
$fh = GetServerFileHandle("myserver");
print $fh "$cmd $argstr\n";
```

It is clear that, while the protocol and back-end allow multiple commands to be sent in a single request, the front end only intends to send a single command. However, the UrlEncode function could leave the "|" character intact. If an attacker provides:

Example Language: *(Attack)*

```
SAY hello world|BAN user12
```

then the front end will see this is a "SAY" command, and the $argstr will look like "hello world | BAN user12". Since the command is "SAY", the check for the "BAN" command will fail, and the front end will send the URL-encoded command to the back end:

Example Language: *(Result)*

```
SAY hello%20world|BAN%20user12
```

The back end, however, will treat these as two separate commands:

Example Language: *(Result)*

```
SAY hello world
BAN user12
```

Notice, however, that if the front end properly encodes the "|" with "%7C", then the back end will only process a single command.

Example 3:

This example takes user input, passes it through an encoding scheme and then creates a directory specified by the user.

Example Language: Perl *(Bad)*

```
sub GetUntrustedInput {
  return($ARGV[0]);
}
sub encode {
  my($str) = @_;
  $str =~ s/\&/\&amp;/gs;
  $str =~ s/\"/\&quot;/gs;
  $str =~ s/\'/\&apos;/gs;
  $str =~ s/\</\&lt;/gs;
  $str =~ s/\>/\&gt;/gs;
  return($str);
}
sub doit {
  my $uname = encode(GetUntrustedInput("username"));
  print "<b>Welcome, $uname!</b><p>\n";
  system("cd /home/$uname; /bin/ls -l");
}
```

The programmer attempts to encode dangerous characters, however the denylist for encoding is incomplete (CWE-184) and an attacker can still pass a semicolon, resulting in a chain with command injection (CWE-77).

Additionally, the encoding routine is used inappropriately with command execution. An attacker doesn't even need to insert their own semicolon. The attacker can instead leverage the encoding routine to provide the semicolon to separate the commands. If an attacker supplies a string of the form:

Example Language: *(Attack)*

```
' pwd
```

then the program will encode the apostrophe and insert the semicolon, which functions as a command separator when passed to the system function. This allows the attacker to complete the command injection.

Observed Examples

Reference	Description
CVE-2021-41232	Chain: authentication routine in Go-based agile development product does not escape user name (CWE-116), allowing LDAP injection (CWE-90) *https://www.cve.org/CVERecord?id=CVE-2021-41232*
CVE-2008-4636	OS command injection in backup software using shell metacharacters in a filename; correct behavior would require that this filename could not be changed. *https://www.cve.org/CVERecord?id=CVE-2008-4636*
CVE-2008-0769	Web application does not set the charset when sending a page to a browser, allowing for XSS exploitation when a browser chooses an unexpected encoding. *https://www.cve.org/CVERecord?id=CVE-2008-0769*
CVE-2008-0005	Program does not set the charset when sending a page to a browser, allowing for XSS exploitation when a browser chooses an unexpected encoding. *https://www.cve.org/CVERecord?id=CVE-2008-0005*
CVE-2008-5573	SQL injection via password parameter; a strong password might contain "&" *https://www.cve.org/CVERecord?id=CVE-2008-5573*
CVE-2008-3773	Cross-site scripting in chat application via a message subject, which normally might contain "&" and other XSS-related characters. *https://www.cve.org/CVERecord?id=CVE-2008-3773*
CVE-2008-0757	Cross-site scripting in chat application via a message, which normally might be allowed to contain arbitrary content. *https://www.cve.org/CVERecord?id=CVE-2008-0757*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	751	2009 Top 25 - Insecure Interaction Between Components	750	2352
MemberOf	C	845	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 2 - Input Validation and Data Sanitization (IDS)	844	2362
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	V	1003	Weaknesses for Simplified Mapping of Published Vulnerabilities	1003	2576
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1179	SEI CERT Perl Coding Standard - Guidelines 01. Input Validation and Data Sanitization (IDS)	1178	2465
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490

Nature	Type	ID	Name	V	Page
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

This weakness is primary to all weaknesses related to injection (CWE-74) since the inherent nature of injection involves the violation of structured messages.

Relationship

CWE-116 and CWE-20 have a close association because, depending on the nature of the structured message, proper input validation can indirectly prevent special characters from changing the meaning of a structured message. For example, by validating that a numeric ID field should only contain the 0-9 characters, the programmer effectively prevents injection attacks. However, input validation is not always sufficient, especially when less stringent data types must be supported, such as free-form text. Consider a SQL injection scenario in which a last name is inserted into a query. The name "O'Reilly" would likely pass the validation step since it is a common last name in the English language. However, it cannot be directly inserted into the database because it contains the "'" apostrophe character, which would need to be escaped or otherwise neutralized. In this case, stripping the apostrophe might reduce the risk of SQL injection, but it would produce incorrect behavior because the wrong name would be recorded.

Terminology

The usage of the "encoding" and "escaping" terms varies widely. For example, in some programming languages, the terms are used interchangeably, while other languages provide APIs that use both terms for different tasks. This overlapping usage extends to the Web, such as the "escape" JavaScript function whose purpose is stated to be encoding. The concepts of encoding and escaping predate the Web by decades. Given such a context, it is difficult for CWE to adopt a consistent vocabulary that will not be misinterpreted by some constituency.

Theoretical

This is a data/directive boundary error in which data boundaries are not sufficiently enforced before it is sent to a different control sphere.

Research Gap

While many published vulnerabilities are related to insufficient output encoding, there is such an emphasis on input validation as a protection mechanism that the underlying causes are rarely described. Within CVE, the focus is primarily on well-understood issues like cross-site scripting and SQL injection. It is likely that this weakness frequently occurs in custom protocols that support multiple encodings, which are not necessarily detectable with automated techniques.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
WASC	22		Improper Output Handling
The CERT Oracle Secure Coding Standard for Java (2011)	IDS00-J	Exact	Sanitize untrusted data passed across a trust boundary
The CERT Oracle Secure Coding Standard for Java (2011)	IDS05-J		Use a subset of ASCII for file and path names
SEI CERT Oracle Coding Standard for Java	IDS00-J	Imprecise	Prevent SQL injection
SEI CERT Perl Coding Standard	IDS33-PL	Exact	Sanitize untrusted data passed across a trust boundary

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
73	User-Controlled Filename
81	Web Server Logs Tampering
85	AJAX Footprinting
104	Cross Zone Scripting

References

[REF-45]OWASP. "OWASP Enterprise Security API (ESAPI) Project". < http://www.owasp.org/index.php/ESAPI >.

[REF-46]Joshbw. "Output Sanitization". 2008 September 8. < https://web.archive.org/web/20081208054333/http://analyticalengine.net/archives/58 >.2023-04-07.

[REF-47]Niyaz PK. "Sanitizing user data: How and where to do it". 2008 September 1. < https://web.archive.org/web/20090105222005/http://www.diovo.com/2008/09/sanitizing-user-data-how-and-where-to-do-it/ >.2023-04-07.

[REF-48]Jeremiah Grossman. "Input validation or output filtering, which is better?". 2007 January 0. < https://blog.jeremiahgrossman.com/2007/01/input-validation-or-output-filtering.html >.2023-04-07.

[REF-49]Jim Manico. "Input Validation - Not That Important". 2008 August 0. < https://manicode.blogspot.com/2008/08/input-validation-not-that-important.html >.2023-04-07.

[REF-50]Michael Eddington. "Preventing XSS with Correct Output Encoding". < http://phed.org/2008/05/19/preventing-xss-with-correct-output-encoding/ >.

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

CWE-117: Improper Output Neutralization for Logs

Weakness ID : 117
Structure : Simple
Abstraction : Base

Description

The product does not neutralize or incorrectly neutralizes output that is written to logs.

Extended Description

This can allow an attacker to forge log entries or inject malicious content into logs.

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.
2. The data is written to an application or system log file.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	116	Improper Encoding or Escaping of Output	281
CanFollow	Ⓑ	93	Improper Neutralization of CRLF Sequences ('CRLF Injection')	217

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1009	Audit	2424

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

Nature	Type	ID	Name	Page
MemberOf	C	1210	Audit / Logging Errors	2475
MemberOf	C	137	Data Neutralization Issues	2311

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	●	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

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

Background Details

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability Non-Repudiation	Modify Application Data Hide Activities Execute Unauthorized Code or Commands *Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. Forged or otherwise corrupted log files can be used to cover an attacker's tracks, possibly by skewing statistics, or even to implicate another party in the commission of a malicious act. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. An attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility.*	

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

Potential Mitigations

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component.

Phase: 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.

Demonstrative Examples

Example 1:

The following web application code attempts to read an integer value from a request object. If the parseInt call fails, then the input is logged with an error message indicating what happened.

Example Language: Java *(Bad)*

```
String val = request.getParameter("val");
try {
  int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
  log.info("Failed to parse val = " + val);
}
...
```

If a user submits the string "twenty-one" for val, the following entry is logged:

- INFO: Failed to parse val=twenty-one

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out %3dbadguy", the following entry is logged:

- INFO: Failed to parse val=twenty-one
- INFO: User logged out=badguy

Clearly, attackers can use this same mechanism to insert arbitrary log entries.

Observed Examples

Reference	Description
CVE-2006-4624	Chain: inject fake log entries with fake timestamps using CRLF injection *https://www.cve.org/CVERecord?id=CVE-2006-4624*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	727	OWASP Top Ten 2004 Category A6 - Injection Flaws	711	2337
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1355	OWASP Top Ten 2021 Category A09:2021 - Security Logging and Monitoring Failures	1344	2496
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Log Forging
Software Fault Patterns	SFP23		Exposed Data
The CERT Oracle Secure Coding Standard for Java (2011)	IDS03-J	Exact	Do not log unsanitized user input
SEI CERT Oracle Coding Standard for Java	IDS03-J	Exact	Do not log unsanitized user input

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
81	Web Server Logs Tampering
93	Log Injection-Tampering-Forging
268	Audit Log Manipulation

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. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

[REF-52]Greg Hoglund and Gary McGraw. "Exploiting Software: How to Break Code". 2004 February 7. Addison-Wesley. < http://www.exploitingsoftware.com/ >.

[REF-53]Alec Muffet. "The night the log was forged". < http://doc.novsu.ac.ru/oreilly/tcpip/puis/ch10_05.htm >.

[REF-43]OWASP. "OWASP TOP 10". 2007 May 8. < https://github.com/owasp-top/owasp-top-2007 >.

CWE-118: Incorrect Access of Indexable Resource ('Range Error')

Weakness ID : 118
Structure : Simple
Abstraction : Class

Description

The product does not restrict or incorrectly restricts operations within the boundaries of a resource that is accessed using an index or pointer, such as memory or files.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454
ParentOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

Applicable Platforms

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

Technology : Not Technology-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Varies by Context	

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP8		Faulty Buffer Access

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
8	Buffer Overflow in an API Call

CAPEC-ID	Attack Pattern Name
9	Buffer Overflow in Local Command-Line Utilities
10	Buffer Overflow via Environment Variables
14	Client-side Injection-induced Buffer Overflow
24	Filter Failure through Buffer Overflow
45	Buffer Overflow via Symbolic Links
46	Overflow Variables and Tags
47	Buffer Overflow via Parameter Expansion

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

Weakness ID : 119
Structure : Simple
Abstraction : Class

Description

The product performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

Extended Description

Certain languages allow direct addressing of memory locations and do not automatically ensure that these locations are valid for the memory buffer that is being referenced. This can cause read or write operations to be performed on memory locations that may be associated with other variables, data structures, or internal program data.

As a result, an attacker may be able to execute arbitrary code, alter the intended control flow, read sensitive information, or cause the system to crash.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	118	Incorrect Access of Indexable Resource ('Range Error')	292
ParentOf	B	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
ParentOf	B	125	Out-of-bounds Read	330
ParentOf	B	466	Return of Pointer Value Outside of Expected Range	1109
ParentOf	B	786	Access of Memory Location Before Start of Buffer	1658
ParentOf	B	787	Out-of-bounds Write	1661
ParentOf	B	788	Access of Memory Location After End of Buffer	1669
ParentOf	B	805	Buffer Access with Incorrect Length Value	1702
ParentOf	B	822	Untrusted Pointer Dereference	1723
ParentOf	B	823	Use of Out-of-range Pointer Offset	1726
ParentOf	B	824	Access of Uninitialized Pointer	1729
ParentOf	B	825	Expired Pointer Dereference	1732
CanFollow	C	20	Improper Input Validation	20
CanFollow	B	128	Wrap-around Error	339

Nature	Type	ID	Name	Page
CanFollow	Ⓥ	129	Improper Validation of Array Index	341
CanFollow	Ⓑ	131	Incorrect Calculation of Buffer Size	355
CanFollow	Ⓑ	190	Integer Overflow or Wraparound	472
CanFollow	Ⓑ	193	Off-by-one Error	486
CanFollow	Ⓥ	195	Signed to Unsigned Conversion Error	494
CanFollow	Ⓑ	839	Numeric Range Comparison Without Minimum Check	1767
CanFollow	Ⓑ	843	Access of Resource Using Incompatible Type ('Type Confusion')	1776
CanFollow	Ⓑ	1257	Improper Access Control Applied to Mirrored or Aliased Memory Regions	2068
CanFollow	Ⓑ	1260	Improper Handling of Overlap Between Protected Memory Ranges	2075
CanFollow	Ⓑ	1339	Insufficient Precision or Accuracy of a Real Number	2242

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ParentOf	Ⓑ	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
ParentOf	Ⓑ	125	Out-of-bounds Read	330
ParentOf	Ⓑ	787	Out-of-bounds Write	1661
ParentOf	Ⓑ	824	Access of Uninitialized Pointer	1729

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

Nature	Type	ID	Name	Page
ParentOf	Ⓑ	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
ParentOf	Ⓑ	123	Write-what-where Condition	323
ParentOf	Ⓑ	125	Out-of-bounds Read	330
ParentOf	Ⓑ	130	Improper Handling of Length Parameter Inconsistency	351
ParentOf	Ⓑ	786	Access of Memory Location Before Start of Buffer	1658
ParentOf	Ⓑ	787	Out-of-bounds Write	1661
ParentOf	Ⓑ	788	Access of Memory Location After End of Buffer	1669
ParentOf	Ⓑ	805	Buffer Access with Incorrect Length Value	1702
ParentOf	Ⓑ	822	Untrusted Pointer Dereference	1723
ParentOf	Ⓑ	823	Use of Out-of-range Pointer Offset	1726
ParentOf	Ⓑ	824	Access of Uninitialized Pointer	1729
ParentOf	Ⓑ	825	Expired Pointer Dereference	1732

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

Nature	Type	ID	Name	Page
ParentOf	Ⓑ	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
ParentOf	Ⓑ	123	Write-what-where Condition	323
ParentOf	Ⓑ	125	Out-of-bounds Read	330
ParentOf	Ⓑ	130	Improper Handling of Length Parameter Inconsistency	351
ParentOf	Ⓑ	786	Access of Memory Location Before Start of Buffer	1658
ParentOf	Ⓑ	787	Out-of-bounds Write	1661
ParentOf	Ⓑ	788	Access of Memory Location After End of Buffer	1669
ParentOf	Ⓑ	805	Buffer Access with Incorrect Length Value	1702
ParentOf	Ⓑ	822	Untrusted Pointer Dereference	1723

Nature	Type	ID	Name	Page
ParentOf	B	823	Use of Out-of-range Pointer Offset	1726
ParentOf	B	824	Access of Uninitialized Pointer	1729
ParentOf	B	825	Expired Pointer Dereference	1732

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Applicable Platforms

Language : C *(Prevalence = Often)*

Language : C++ *(Prevalence = Often)*

Language : Assembly *(Prevalence = Undetermined)*

Alternate Terms

Buffer Overflow : This term has many different meanings to different audiences. From a CWE mapping perspective, this term should be avoided where possible. Some researchers, developers, and tools intend for it to mean "write past the end of a buffer," whereas others use the same term to mean "any read or write outside the boundaries of a buffer, whether before the beginning of the buffer or after the end of the buffer." Still others using the same term could mean "any action after the end of a buffer, whether it is a read or write." Since the term is commonly used for exploitation and for vulnerabilities, it further confuses things.

buffer overrun : Some prominent vendors and researchers use the term "buffer overrun," but most people use "buffer overflow." See the alternate term for "buffer overflow" for context.

memory safety : Generally used for techniques that avoid weaknesses related to memory access, such as those identified by CWE-119 and its descendants. However, the term is not formal, and there is likely disagreement between practitioners as to which weaknesses are implicitly covered by the "memory safety" term.

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands Modify Memory *If the memory accessible by the attacker can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow. If the attacker can overwrite a pointer's worth of memory (usually 32 or 64 bits), they can redirect a function pointer to their own malicious code. Even when the attacker can only modify a single byte arbitrary code execution can be possible. Sometimes this is because the same problem can be exploited repeatedly to the same effect. Other times it is because the attacker can overwrite security-critical application-specific data -- such as a flag indicating whether the user is an administrator.*	
Availability Confidentiality	Read Memory DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory)	

Scope	Impact	Likelihood
	Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.	
Confidentiality	Read Memory *In the case of an out-of-bounds read, the attacker may have access to sensitive information. If the sensitive information contains system details, such as the current buffers position in memory, this knowledge can be used to craft further attacks, possibly with more severe consequences.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives. Automated static analysis generally does not account for environmental considerations when reporting out-of-bounds memory operations. This can make it difficult for users to determine which warnings should be investigated first. For example, an analysis tool might report buffer overflows that originate from command line arguments in a program that is not expected to run with setuid or other special privileges.

Effectiveness = High

Detection techniques for buffer-related errors are more mature than for most other weakness types.

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode Quality Analysis 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: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies

Effectiveness = SOAR Partial

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Web Application Scanner Web Services Scanner Database Scanners

Effectiveness = SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Fuzz Tester Framework-based Fuzzer

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source Manual Source Code Review (not inspections)

Effectiveness = SOAR Partial

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 Cost effective for partial coverage: Source Code Quality Analyzer

Effectiveness = High

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Requirements

Strategy = Language Selection

Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer. Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples include the Safe C String Library (SafeStr) by Messier and Viega [REF-57], and the Strsafe.h library from Microsoft [REF-56]. These libraries provide safer versions of overflow-prone string-handling functions.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Use automatic buffer overflow detection mechanisms that are offered by certain compilers or compiler extensions. Examples include: the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND [REF-1334] discusses canary-based detection in detail.

Effectiveness = Defense in Depth

This is not necessarily a complete solution, since these mechanisms only detect certain types of overflows. In addition, the result is still a denial of service, since the typical response is to exit the application.

Phase: Implementation

Consider adhering to the following rules when allocating and managing an application's memory: Double check that the buffer is as large as specified. When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string. Check buffer boundaries if accessing the buffer in a loop and make sure there is no danger of writing past the allocated space. If necessary, truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335].

Effectiveness = Defense in Depth

These techniques do not provide a complete solution. For instance, exploits frequently use a bug that discloses memory addresses in order to maximize reliability of code execution [REF-1337]. It has also been shown that a side-channel attack can bypass ASLR [REF-1333]

Phase: Operation

Strategy = Environment Hardening

Use a CPU and operating system that offers Data Execution Protection (using hardware NX or XD bits) or the equivalent techniques that simulate this feature in software, such as PaX [REF-60] [REF-61]. These techniques ensure that any instruction executed is exclusively at a memory address that is part of the code segment. For more information on these techniques see D3-PSEP (Process Segment Execution Prevention) from D3FEND [REF-1336].

Effectiveness = Defense in Depth

This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Phase: Implementation

Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.

Effectiveness = Moderate

This approach is still susceptible to calculation errors, including issues such as off-by-one errors (CWE-193) and incorrectly calculating buffer lengths (CWE-131).

Demonstrative Examples

Example 1:

This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.

Example Language: C *(Bad)*

```
void host_lookup(char *user_supplied_addr){
  struct hostent *hp;
  in_addr_t *addr;
  char hostname[64];
  in_addr_t inet_addr(const char *cp);
  /*routine that ensures user_supplied_addr is in the right format for conversion */
  validate_addr_form(user_supplied_addr);
  addr = inet_addr(user_supplied_addr);
  hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
  strcpy(hostname, hp->h_name);
}
```

This function allocates a buffer of 64 bytes to store the hostname, however there is no guarantee that the hostname will not be larger than 64 bytes. If an attacker specifies an address which resolves to a very large hostname, then the function may overwrite sensitive data or even relinquish control flow to the attacker.

Note that this example also contains an unchecked return value (CWE-252) that can lead to a NULL pointer dereference (CWE-476).

Example 2:

This example applies an encoding procedure to an input string and stores it into a buffer.

Example Language: C *(Bad)*

```
char * copy_input(char *user_supplied_string){
  int i, dst_index;
  char *dst_buf = (char*)malloc(4*sizeof(char) * MAX_SIZE);
  if ( MAX_SIZE <= strlen(user_supplied_string) ){
    die("user string too long, die evil hacker!");
  }
  dst_index = 0;
  for ( i = 0; i < strlen(user_supplied_string); i++ ){
    if( '&' == user_supplied_string[i] ){
      dst_buf[dst_index++] = '&';
      dst_buf[dst_index++] = 'a';
      dst_buf[dst_index++] = 'm';
      dst_buf[dst_index++] = 'p';
      dst_buf[dst_index++] = ';';
    }
    else if ('<' == user_supplied_string[i] ){
      /* encode to &lt; */
    }
    else dst_buf[dst_index++] = user_supplied_string[i];
  }
  return dst_buf;
}
```

The programmer attempts to encode the ampersand character in the user-controlled string, however the length of the string is validated before the encoding procedure is applied. Furthermore, the programmer assumes encoding expansion will only expand a given character by a factor of 4, while the encoding of the ampersand expands by 5. As a result, when the encoding procedure expands the string it is possible to overflow the destination buffer if the attacker provides a string of many ampersands.

Example 3:

The following example asks a user for an offset into an array to select an item.

Example Language: C *(Bad)*

```
int main (int argc, char **argv) {
  char *items[] = {"boat", "car", "truck", "train"};
```

```
   int index = GetUntrustedOffset();
   printf("You selected %s\n", items[index-1]);
}
```

The programmer allows the user to specify which element in the list to select, however an attacker can provide an out-of-bounds offset, resulting in a buffer over-read (CWE-126).

Example 4:

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method

Example Language: C *(Bad)*

```
int getValueFromArray(int *array, int len, int index) {
   int value;
   // check that the array index is less than the maximum
   // length of the array
   if (index < len) {
      // get the value at the specified index of the array
      value = array[index];
   }
   // if array index is invalid then output error message
   // and return value indicating error
   else {
      printf("Value is: %d\n", array[index]);
      value = -1;
   }
   return value;
}
```

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

Example Language: C *(Good)*

```
...
// check that the array index is within the correct
// range of values for the array
if (index >= 0 && index < len) {
...
```

Example 5:

Windows provides the _mbs family of functions to perform various operations on multibyte strings. When these functions are passed a malformed multibyte string, such as a string containing a valid leading byte followed by a single null byte, they can read or write past the end of the string buffer causing a buffer overflow. The following functions all pose a risk of buffer overflow: _mbsinc _mbsdec _mbsncat _mbsncpy _mbsnextc _mbsnset _mbsrev _mbsset _mbsstr _mbstok _mbccpy _mbslen

Observed Examples

Reference	Description
CVE-2021-22991	Incorrect URI normalization in application traffic product leads to buffer overflow, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2021-22991*
CVE-2020-29557	Buffer overflow in Wi-Fi router web interface, as exploited in the wild per CISA KEV.

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2020-29557
CVE-2009-2550	Classic stack-based buffer overflow in media player using a long entry in a playlist *https://www.cve.org/CVERecord?id=CVE-2009-2550*
CVE-2009-2403	Heap-based buffer overflow in media player using a long entry in a playlist *https://www.cve.org/CVERecord?id=CVE-2009-2403*
CVE-2009-0689	large precision value in a format string triggers overflow *https://www.cve.org/CVERecord?id=CVE-2009-0689*
CVE-2009-0690	negative offset value leads to out-of-bounds read *https://www.cve.org/CVERecord?id=CVE-2009-0690*
CVE-2009-1532	malformed inputs cause accesses of uninitialized or previously-deleted objects, leading to memory corruption *https://www.cve.org/CVERecord?id=CVE-2009-1532*
CVE-2009-1528	chain: lack of synchronization leads to memory corruption *https://www.cve.org/CVERecord?id=CVE-2009-1528*
CVE-2021-29529	Chain: machine-learning product can have a heap-based buffer overflow (CWE-122) when some integer-oriented bounds are calculated by using ceiling() and floor() on floating point values (CWE-1339) *https://www.cve.org/CVERecord?id=CVE-2021-29529*
CVE-2009-0558	attacker-controlled array index leads to code execution *https://www.cve.org/CVERecord?id=CVE-2009-0558*
CVE-2009-0269	chain: -1 value from a function call was intended to indicate an error, but is used as an array index instead. *https://www.cve.org/CVERecord?id=CVE-2009-0269*
CVE-2009-0566	chain: incorrect calculations lead to incorrect pointer dereference and memory corruption *https://www.cve.org/CVERecord?id=CVE-2009-0566*
CVE-2009-1350	product accepts crafted messages that lead to a dereference of an arbitrary pointer *https://www.cve.org/CVERecord?id=CVE-2009-1350*
CVE-2009-0191	chain: malformed input causes dereference of uninitialized memory *https://www.cve.org/CVERecord?id=CVE-2009-0191*
CVE-2008-4113	OS kernel trusts userland-supplied length value, allowing reading of sensitive information *https://www.cve.org/CVERecord?id=CVE-2008-4113*
CVE-2005-1513	Chain: integer overflow in securely-coded mail program leads to buffer overflow. In 2005, this was regarded as unrealistic to exploit, but in 2020, it was rediscovered to be easier to exploit due to evolutions of the technology. *https://www.cve.org/CVERecord?id=CVE-2005-1513*
CVE-2003-0542	buffer overflow involving a regular expression with a large number of captures *https://www.cve.org/CVERecord?id=CVE-2003-0542*
CVE-2017-1000121	chain: unchecked message size metadata allows integer overflow (CWE-190) leading to buffer overflow (CWE-119). *https://www.cve.org/CVERecord?id=CVE-2017-1000121*

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	635	Weaknesses Originally Used by NVD from 2008 to 2016	635	2552
MemberOf	C	726	OWASP Top Ten 2004 Category A5 - Buffer Overflows	711	2336
MemberOf	C	740	CERT C Secure Coding Standard (2008) Chapter 7 - Arrays (ARR)	734	2344
MemberOf	C	741	CERT C Secure Coding Standard (2008) Chapter 8 - Characters and Strings (STR)	734	2344
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	743	CERT C Secure Coding Standard (2008) Chapter 10 - Input Output (FIO)	734	2347
MemberOf	C	744	CERT C Secure Coding Standard (2008) Chapter 11 - Environment (ENV)	734	2348
MemberOf	C	752	2009 Top 25 - Risky Resource Management	750	2353
MemberOf	C	874	CERT C++ Secure Coding Section 06 - Arrays and the STL (ARR)	868	2375
MemberOf	C	875	CERT C++ Secure Coding Section 07 - Characters and Strings (STR)	868	2376
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	C	877	CERT C++ Secure Coding Section 09 - Input Output (FIO)	868	2377
MemberOf	C	878	CERT C++ Secure Coding Section 10 - Environment (ENV)	868	2378
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	V	1003	Weaknesses for Simplified Mapping of Published Vulnerabilities	1003	2576
MemberOf	C	1157	SEI CERT C Coding Standard - Guidelines 03. Expressions (EXP)	1154	2455
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	1387	2597
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses	1425	2600

Notes

Applicable Platform

It is possible in any programming languages without memory management support to attempt an operation outside of the bounds of a memory buffer, but the consequences will vary widely depending on the language, platform, and chip architecture.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A5	Exact	Buffer Overflows
CERT C Secure Coding	ARR00-C		Understand how arrays work
CERT C Secure Coding	ARR30-C	CWE More Abstract	Do not form or use out-of-bounds pointers or array subscripts
CERT C Secure Coding	ARR38-C	CWE More Abstract	Guarantee that library functions do not form invalid pointers
CERT C Secure Coding	ENV01-C		Do not make assumptions about the size of an environment variable
CERT C Secure Coding	EXP39-C	Imprecise	Do not access a variable through a pointer of an incompatible type
CERT C Secure Coding	FIO37-C		Do not assume character data has been read
CERT C Secure Coding	STR31-C	CWE More Abstract	Guarantee that storage for strings has sufficient space for character data and the null terminator
CERT C Secure Coding	STR32-C	CWE More Abstract	Do not pass a non-null-terminated character sequence to a library function that expects a string
WASC	7		Buffer Overflow
Software Fault Patterns	SFP8		Faulty Buffer Access

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
8	Buffer Overflow in an API Call
9	Buffer Overflow in Local Command-Line Utilities
10	Buffer Overflow via Environment Variables
14	Client-side Injection-induced Buffer Overflow
24	Filter Failure through Buffer Overflow
42	MIME Conversion
44	Overflow Binary Resource File
45	Buffer Overflow via Symbolic Links
46	Overflow Variables and Tags
47	Buffer Overflow via Parameter Expansion
100	Overflow Buffers
123	Buffer Manipulation

References

[REF-1029]Aleph One. "Smashing The Stack For Fun And Profit". 1996 November 8. < http://phrack.org/issues/49/14.html >.

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-56]Microsoft. "Using the Strsafe.h Functions". < https://learn.microsoft.com/en-us/windows/win32/menurc/strsafe-ovw?redirectedfrom=MSDN >.2023-04-07.

[REF-57]Matt Messier and John Viega. "Safe C String Library v1.0.3". < http://www.gnu-darwin.org/www001/ports-1.5a-CURRENT/devel/safestr/work/safestr-1.0.3/doc/safestr.html >.2023-04-07.

[REF-58]Michael Howard. "Address Space Layout Randomization in Windows Vista". < https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista >.2023-04-07.

[REF-59]Arjan van de Ven. "Limiting buffer overflows with ExecShield". < https://archive.is/saAFo >.2023-04-07.

[REF-60]"PaX". < https://en.wikipedia.org/wiki/Executable_space_protection#PaX >.2023-04-07.

[REF-61]Microsoft. "Understanding DEP as a mitigation technology part 1". < https://msrc.microsoft.com/blog/2009/06/understanding-dep-as-a-mitigation-technology-part-1/ >.2023-04-07.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-64]Grant Murphy. "Position Independent Executables (PIE)". 2012 November 8. Red Hat. < https://www.redhat.com/en/blog/position-independent-executables-pie >.2023-04-07.

[REF-1332]John Richard Moser. "Prelink and address space randomization". 2006 July 5. < https://lwn.net/Articles/190139/ >.2023-04-26.

[REF-1333]Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". 2016. < http://www.cs.ucr.edu/~nael/pubs/micro16.pdf >.2023-04-26.

[REF-1334]D3FEND. "Stack Frame Canary Validation (D3-SFCV)". 2023. < https://d3fend.mitre.org/technique/d3f:StackFrameCanaryValidation/ >.2023-04-26.

[REF-1335]D3FEND. "Segment Address Offset Randomization (D3-SAOR)". 2023. < https://d3fend.mitre.org/technique/d3f:SegmentAddressOffsetRandomization/ >.2023-04-26.

[REF-1336]D3FEND. "Process Segment Execution Prevention (D3-PSEP)". 2023. < https://d3fend.mitre.org/technique/d3f:ProcessSegmentExecutionPrevention/ >.2023-04-26.

[REF-1337]Alexander Sotirov and Mark Dowd. "Bypassing Browser Memory Protections: Setting back browser security by 10 years". 2008. < https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf >.2023-04-26.

CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Weakness ID : 120
Structure : Simple
Abstraction : Base

Description

The product copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

Extended Description

A buffer overflow condition exists when a product attempts to put more data in a buffer than it can hold, or when it attempts to put data in a memory area outside of the boundaries of a buffer. The simplest type of error, and the most common cause of buffer overflows, is the "classic" case in which the product copies the buffer without restricting how much is copied. Other variants exist, but the existence of a classic overflow strongly suggests that the programmer is not considering even the most basic of security protections.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293
ParentOf	V	785	Use of Path Manipulation Function without Maximum-sized Buffer	1656
CanFollow	B	170	Improper Null Termination	428
CanFollow	V	231	Improper Handling of Extra Values	572
CanFollow	V	416	Use After Free	1012
CanFollow	V	456	Missing Initialization of a Variable	1089
CanPrecede	B	123	Write-what-where Condition	323

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
MemberOf	C	1218	Memory Buffer Errors	2479

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Resultant :

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Assembly *(Prevalence = Undetermined)*

Alternate Terms

Classic Buffer Overflow : This term was frequently used by vulnerability researchers during approximately 1995 to 2005 to differentiate buffer copies without length checks (which had been known about for decades) from other emerging weaknesses that still involved invalid accesses of buffers, as vulnerability researchers began to develop advanced exploitation techniques.

Unbounded Transfer :

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Modify Memory Execute Unauthorized Code or Commands *Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of the product's implicit security policy. This can often be used to subvert any other security service.*	
Availability	Modify Memory DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) *Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the product into an infinite loop.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives. Automated static analysis generally does not account for environmental considerations when reporting out-of-bounds memory operations. This can make it difficult for users to determine which warnings should be investigated first. For example, an analysis tool might report buffer overflows that originate from command line arguments in a program that is not expected to run with setuid or other special privileges.

Effectiveness = High

Detection techniques for buffer-related errors are more mature than for most other weakness types.

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Manual Analysis

Manual analysis can be useful for finding this weakness, but it might not achieve desired code coverage within limited time constraints. This becomes difficult for weaknesses that must be considered for all inputs, since the attack surface can be too large.

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Bytecode Weakness Analysis - including disassembler + source code weakness analysis Binary Weakness Analysis - including disassembler + source code weakness analysis

Effectiveness = High

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies

Effectiveness = SOAR Partial

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Web Application Scanner Web Services Scanner Database Scanners

Effectiveness = SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Fuzz Tester Framework-based Fuzzer

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source Manual Source Code Review (not inspections)

Effectiveness = SOAR Partial

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

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Requirements

Strategy = Language Selection

Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer. Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples include the Safe C String Library (SafeStr) by Messier and Viega [REF-57], and the Strsafe.h library from Microsoft [REF-56]. These libraries provide safer versions of overflow-prone string-handling functions.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Use automatic buffer overflow detection mechanisms that are offered by certain compilers or compiler extensions. Examples include: the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND [REF-1334] discusses canary-based detection in detail.

Effectiveness = Defense in Depth

This is not necessarily a complete solution, since these mechanisms only detect certain types of overflows. In addition, the result is still a denial of service, since the typical response is to exit the application.

Phase: Implementation

Consider adhering to the following rules when allocating and managing an application's memory: Double check that your buffer is as large as you specify. When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string. Check buffer boundaries if accessing the buffer in a loop and make sure there is no danger of writing past the allocated space. If necessary, truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.

Phase: 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.

Phase: 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.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335].

Effectiveness = Defense in Depth

These techniques do not provide a complete solution. For instance, exploits frequently use a bug that discloses memory addresses in order to maximize reliability of code execution [REF-1337]. It has also been shown that a side-channel attack can bypass ASLR [REF-1333]

Phase: Operation

Strategy = Environment Hardening

Use a CPU and operating system that offers Data Execution Protection (using hardware NX or XD bits) or the equivalent techniques that simulate this feature in software, such as PaX [REF-60] [REF-61]. These techniques ensure that any instruction executed is exclusively at a memory address that is part of the code segment. For more information on these techniques see D3-PSEP (Process Segment Execution Prevention) from D3FEND [REF-1336].

Effectiveness = Defense in Depth

This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Phase: Build and Compilation

Phase: Operation

Most mitigating technologies at the compiler or OS level to date address only a subset of buffer overflow problems and rarely provide complete protection against even that subset. It is good practice to implement strategies to increase the workload of an attacker, such as leaving the attacker to guess an unknown value that changes every program execution.

Phase: Implementation

Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.

Effectiveness = Moderate

This approach is still susceptible to calculation errors, including issues such as off-by-one errors (CWE-193) and incorrectly calculating buffer lengths (CWE-131).

Phase: Architecture and Design

Strategy = Enforcement by Conversion

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.

Phase: Architecture and Design

Phase: Operation

Strategy = Environment Hardening

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.

Phase: Architecture and Design

Phase: Operation

Strategy = Sandbox or Jail

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

The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Demonstrative Examples

Example 1:

The following code asks the user to enter their last name and then attempts to store the value entered in the last_name array.

Example Language: C *(Bad)*

```
char last_name[20];
printf ("Enter your last name: ");
scanf ("%s", last_name);
```

The problem with the code above is that it does not restrict or limit the size of the name entered by the user. If the user enters "Very_very_long_last_name" which is 24 characters long, then a buffer overflow will occur since the array can only hold 20 characters total.

Example 2:

The following code attempts to create a local copy of a buffer to perform some manipulations to the data.

Example Language: C *(Bad)*

```
void manipulate_string(char * string){
  char buf[24];
  strcpy(buf, string);
  ...
}
```

However, the programmer does not ensure that the size of the data pointed to by string will fit in the local buffer and copies the data with the potentially dangerous strcpy() function. This may result in a buffer overflow condition if an attacker can influence the contents of the string parameter.

Example 3:

The code below calls the gets() function to read in data from the command line.

Example Language: C *(Bad)*

```
  char buf[24];
  printf("Please enter your name and press <Enter>\n");
  gets(buf);
  ...
}
```

However, gets() is inherently unsafe, because it copies all input from STDIN to the buffer without checking size. This allows the user to provide a string that is larger than the buffer size, resulting in an overflow condition.

Example 4:

In the following example, a server accepts connections from a client and processes the client request. After accepting a client connection, the program will obtain client information using the gethostbyaddr method, copy the hostname of the client that connected to a local variable and output the hostname of the client to a log file.

Example Language: C *(Bad)*

```
...
  struct hostent *clienthp;
  char hostname[MAX_LEN];
  // create server socket, bind to server address and listen on socket
  ...
  // accept client connections and process requests
  int count = 0;
  for (count = 0; count < MAX_CONNECTIONS; count++) {
    int clientlen = sizeof(struct sockaddr_in);
    int clientsocket = accept(serversocket, (struct sockaddr *)&clientaddr, &clientlen);
    if (clientsocket >= 0) {
      clienthp = gethostbyaddr((char*) &clientaddr.sin_addr.s_addr, sizeof(clientaddr.sin_addr.s_addr), AF_INET);
      strcpy(hostname, clienthp->h_name);
      logOutput("Accepted client connection from host ", hostname);
      // process client request
      ...
      close(clientsocket);
    }
  }
  close(serversocket);
...
```

However, the hostname of the client that connected may be longer than the allocated size for the local hostname variable. This will result in a buffer overflow when copying the client hostname to the local variable using the strcpy method.

Observed Examples

Reference	Description
CVE-2000-1094	buffer overflow using command with long argument *https://www.cve.org/CVERecord?id=CVE-2000-1094*
CVE-1999-0046	buffer overflow in local program using long environment variable *https://www.cve.org/CVERecord?id=CVE-1999-0046*
CVE-2002-1337	buffer overflow in comment characters, when product increments a counter for a ">" but does not decrement for "<" *https://www.cve.org/CVERecord?id=CVE-2002-1337*
CVE-2003-0595	By replacing a valid cookie value with an extremely long string of characters, an attacker may overflow the application's buffers. *https://www.cve.org/CVERecord?id=CVE-2003-0595*
CVE-2001-0191	By replacing a valid cookie value with an extremely long string of characters, an attacker may overflow the application's buffers. *https://www.cve.org/CVERecord?id=CVE-2001-0191*

Functional Areas

- Memory Management

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334

Nature	Type	ID	Name	V	Page
MemberOf	C	726	OWASP Top Ten 2004 Category A5 - Buffer Overflows	711	2336
MemberOf	C	741	CERT C Secure Coding Standard (2008) Chapter 8 - Characters and Strings (STR)	734	2344
MemberOf	C	802	2010 Top 25 - Risky Resource Management	800	2354
MemberOf	C	865	2011 Top 25 - Risky Resource Management	900	2371
MemberOf	C	875	CERT C++ Secure Coding Section 07 - Characters and Strings (STR)	868	2376
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1129	CISQ Quality Measures (2016) - Reliability	1128	2440
MemberOf	C	1131	CISQ Quality Measures (2016) - Security	1128	2442
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Relationship

At the code level, stack-based and heap-based overflows do not differ significantly, so there usually is not a need to distinguish them. From the attacker perspective, they can be quite different, since different techniques are required to exploit them.

Terminology

Many issues that are now called "buffer overflows" are substantively different than the "classic" overflow, including entirely different bug types that rely on overflow exploit techniques, such as integer signedness errors, integer overflows, and format string bugs. This imprecise terminology can make it difficult to determine which variant is being reported.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Unbounded Transfer ('classic overflow')
7 Pernicious Kingdoms			Buffer Overflow
CLASP			Buffer overflow
OWASP Top Ten 2004	A1	CWE More Specific	Unvalidated Input
OWASP Top Ten 2004	A5	CWE More Specific	Buffer Overflows
CERT C Secure Coding	STR31-C	Exact	Guarantee that storage for strings has sufficient space for character data and the null terminator
WASC	7		Buffer Overflow
Software Fault Patterns	SFP8		Faulty Buffer Access
OMG ASCSM	ASCSM-CWE-120		
OMG ASCRM	ASCRM-CWE-120		

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
8	Buffer Overflow in an API Call
9	Buffer Overflow in Local Command-Line Utilities
10	Buffer Overflow via Environment Variables
14	Client-side Injection-induced Buffer Overflow
24	Filter Failure through Buffer Overflow
42	MIME Conversion

CAPEC-ID	Attack Pattern Name
44	Overflow Binary Resource File
45	Buffer Overflow via Symbolic Links
46	Overflow Variables and Tags
47	Buffer Overflow via Parameter Expansion
67	String Format Overflow in syslog()
92	Forced Integer Overflow
100	Overflow Buffers

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-56]Microsoft. "Using the Strsafe.h Functions". < https://learn.microsoft.com/en-us/windows/win32/menurc/strsafe-ovw?redirectedfrom=MSDN >.2023-04-07.

[REF-57]Matt Messier and John Viega. "Safe C String Library v1.0.3". < http://www.gnu-darwin.org/www001/ports-1.5a-CURRENT/devel/safestr/work/safestr-1.0.3/doc/safestr.html >.2023-04-07.

[REF-58]Michael Howard. "Address Space Layout Randomization in Windows Vista". < https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista >.2023-04-07.

[REF-59]Arjan van de Ven. "Limiting buffer overflows with ExecShield". < https://archive.is/saAFo >.2023-04-07.

[REF-60]"PaX". < https://en.wikipedia.org/wiki/Executable_space_protection#PaX >.2023-04-07.

[REF-74]Jason Lam. "Top 25 Series - Rank 3 - Classic Buffer Overflow". 2010 March 2. SANS Software Security Institute. < http://software-security.sans.org/blog/2010/03/02/top-25-series-rank-3-classic-buffer-overflow/ >.

[REF-61]Microsoft. "Understanding DEP as a mitigation technology part 1". < https://msrc.microsoft.com/blog/2009/06/understanding-dep-as-a-mitigation-technology-part-1/ >.2023-04-07.

[REF-76]Sean Barnum and Michael Gegick. "Least Privilege". 2005 September 4. < https://web.archive.org/web/20211209014121/https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege >.2023-04-07.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-64]Grant Murphy. "Position Independent Executables (PIE)". 2012 November 8. Red Hat. < https://www.redhat.com/en/blog/position-independent-executables-pie >.2023-04-07.

[REF-961]Object Management Group (OMG). "Automated Source Code Reliability Measure (ASCRM)". 2016 January. < http://www.omg.org/spec/ASCRM/1.0/ >.

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

[REF-1332]John Richard Moser. "Prelink and address space randomization". 2006 July 5. < https://lwn.net/Articles/190139/ >.2023-04-26.

[REF-1333]Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". 2016. < http://www.cs.ucr.edu/~nael/pubs/micro16.pdf >.2023-04-26.

[REF-1334]D3FEND. "Stack Frame Canary Validation (D3-SFCV)". 2023. < https://d3fend.mitre.org/technique/d3f:StackFrameCanaryValidation/ >.2023-04-26.

[REF-1335]D3FEND. "Segment Address Offset Randomization (D3-SAOR)". 2023. < https://d3fend.mitre.org/technique/d3f:SegmentAddressOffsetRandomization/ >.2023-04-26.

[REF-1336]D3FEND. "Process Segment Execution Prevention (D3-PSEP)". 2023. < https://d3fend.mitre.org/technique/d3f:ProcessSegmentExecutionPrevention/ >.2023-04-26.

[REF-1337]Alexander Sotirov and Mark Dowd. "Bypassing Browser Memory Protections: Setting back browser security by 10 years". 2008. < https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf >.2023-04-26.

CWE-121: Stack-based Buffer Overflow

Weakness ID : 121
Structure : Simple
Abstraction : Variant

Description

A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	787	Out-of-bounds Write	1661
ChildOf	B	788	Access of Memory Location After End of Buffer	1669

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Background Details

There are generally several security-critical data on an execution stack that can lead to arbitrary code execution. The most prominent is the stored return address, the memory address at which execution should continue once the current function is finished executing. The attacker can overwrite this value with some memory address to which the attacker also has write access, into which they place arbitrary code to be run with the full privileges of the vulnerable program. Alternately, the attacker can supply the address of an important call, for instance the POSIX system() call, leaving arguments to the call on the stack. This is often called a return into libc exploit, since the attacker generally forces the program to jump at return time into an interesting routine in the C standard library (libc). Other important data commonly on the stack include the stack pointer and frame pointer, two values that indicate offsets for computing memory addresses. Modifying those values can often be leveraged into a "write-what-where" condition.

Alternate Terms

Stack Overflow : "Stack Overflow" is often used to mean the same thing as stack-based buffer overflow, however it is also used on occasion to mean stack exhaustion, usually a result from an excessively recursive function call. Due to the ambiguity of the term, use of stack overflow to describe either circumstance is discouraged.

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Availability	Modify Memory DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) *Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.*	
Integrity Confidentiality Availability Access Control	Modify Memory Execute Unauthorized Code or Commands Bypass Protection Mechanism *Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.*	
Integrity Confidentiality Availability Access Control Other	Modify Memory Execute Unauthorized Code or Commands Bypass Protection Mechanism Other *When the consequence is arbitrary code execution, this can often be used to subvert any other security service.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

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

Potential Mitigations

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Use automatic buffer overflow detection mechanisms that are offered by certain compilers or compiler extensions. Examples include: the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND [REF-1334] discusses canary-based detection in detail.

Effectiveness = Defense in Depth

This is not necessarily a complete solution, since these mechanisms only detect certain types of overflows. In addition, the result is still a denial of service, since the typical response is to exit the application.

Phase: Architecture and Design

Use an abstraction library to abstract away risky APIs. Not a complete solution.

Phase: Implementation

Implement and perform bounds checking on input.

Phase: Implementation

Do not use dangerous functions such as gets. Use safer, equivalent functions which check for boundary errors.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335].

Effectiveness = Defense in Depth

These techniques do not provide a complete solution. For instance, exploits frequently use a bug that discloses memory addresses in order to maximize reliability of code execution [REF-1337]. It has also been shown that a side-channel attack can bypass ASLR [REF-1333]

Demonstrative Examples

Example 1:

While buffer overflow examples can be rather complex, it is possible to have very simple, yet still exploitable, stack-based buffer overflows:

Example Language: C *(Bad)*

```
#define BUFSIZE 256
int main(int argc, char **argv) {
  char buf[BUFSIZE];
  strcpy(buf, argv[1]);
}
```

The buffer size is fixed, but there is no guarantee the string in argv[1] will not exceed this size and cause an overflow.

Example 2:

This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.

Example Language: C *(Bad)*

```
void host_lookup(char *user_supplied_addr){
  struct hostent *hp;
  in_addr_t *addr;
  char hostname[64];
  in_addr_t inet_addr(const char *cp);
  /*routine that ensures user_supplied_addr is in the right format for conversion */
  validate_addr_form(user_supplied_addr);
  addr = inet_addr(user_supplied_addr);
  hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
  strcpy(hostname, hp->h_name);
}
```

This function allocates a buffer of 64 bytes to store the hostname, however there is no guarantee that the hostname will not be larger than 64 bytes. If an attacker specifies an address which resolves to a very large hostname, then the function may overwrite sensitive data or even relinquish control flow to the attacker.

Note that this example also contains an unchecked return value (CWE-252) that can lead to a NULL pointer dereference (CWE-476).

Observed Examples

Reference	Description
CVE-2021-35395	Stack-based buffer overflows in SFK for wifi chipset used for IoT/embedded devices, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2021-35395*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	C	1365	ICS Communications: Unreliability	1358	2502
MemberOf	C	1366	ICS Communications: Frail Security in Protocols	1358	2503
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Other

Stack-based buffer overflows can instantiate in return address overwrites, stack pointer overwrites or frame pointer overwrites. They can also be considered function pointer overwrites, array indexer overwrites or write-what-where condition, etc.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Stack overflow
Software Fault Patterns	SFP8		Faulty Buffer Access

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	ARR38-C	Imprecise	Guarantee that library functions do not form invalid pointers
CERT C Secure Coding	STR31-C	CWE More Specific	Guarantee that storage for strings has sufficient space for character data and the null terminator

References

[REF-1029]Aleph One. "Smashing The Stack For Fun And Profit". 1996 November 8. < http://phrack.org/issues/49/14.html >.

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-58]Michael Howard. "Address Space Layout Randomization in Windows Vista". < https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista >.2023-04-07.

[REF-60]"PaX". < https://en.wikipedia.org/wiki/Executable_space_protection#PaX >.2023-04-07.

[REF-64]Grant Murphy. "Position Independent Executables (PIE)". 2012 November 8. Red Hat. < https://www.redhat.com/en/blog/position-independent-executables-pie >.2023-04-07.

[REF-1332]John Richard Moser. "Prelink and address space randomization". 2006 July 5. < https://lwn.net/Articles/190139/ >.2023-04-26.

[REF-1333]Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". 2016. < http://www.cs.ucr.edu/~nael/pubs/micro16.pdf >.2023-04-26.

[REF-1334]D3FEND. "Stack Frame Canary Validation (D3-SFCV)". 2023. < https://d3fend.mitre.org/technique/d3f:StackFrameCanaryValidation/ >.2023-04-26.

[REF-1335]D3FEND. "Segment Address Offset Randomization (D3-SAOR)". 2023. < https://d3fend.mitre.org/technique/d3f:SegmentAddressOffsetRandomization/ >.2023-04-26.

[REF-1337]Alexander Sotirov and Mark Dowd. "Bypassing Browser Memory Protections: Setting back browser security by 10 years". 2008. < https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf >.2023-04-26.

CWE-122: Heap-based Buffer Overflow

Weakness ID : 122
Structure : Simple
Abstraction : Variant

Description

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as malloc().

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	787	Out-of-bounds Write	1661
ChildOf	Ⓑ	788	Access of Memory Location After End of Buffer	1669

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) *Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.*	
Integrity Confidentiality Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism Modify Memory *Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. Besides important user data, heap-based overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually leave many in memory. For example, object methods in C++ are generally implemented using function pointers. Even in C programs, there is often a global offset table used by the underlying runtime.*	
Integrity Confidentiality Availability Access Control Other	Execute Unauthorized Code or Commands Bypass Protection Mechanism Other *When the consequence is arbitrary code execution, this can often be used to subvert any other security service.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Potential Mitigations

Pre-design: Use a language or compiler that performs automatic bounds checking.

Phase: Architecture and Design

Use an abstraction library to abstract away risky APIs. Not a complete solution.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Use automatic buffer overflow detection mechanisms that are offered by certain compilers or compiler extensions. Examples include: the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND [REF-1334] discusses canary-based detection in detail.

Effectiveness = Defense in Depth

This is not necessarily a complete solution, since these mechanisms only detect certain types of overflows. In addition, the result is still a denial of service, since the typical response is to exit the application.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335].

Effectiveness = Defense in Depth

These techniques do not provide a complete solution. For instance, exploits frequently use a bug that discloses memory addresses in order to maximize reliability of code execution [REF-1337]. It has also been shown that a side-channel attack can bypass ASLR [REF-1333]

Phase: Implementation

Implement and perform bounds checking on input.

Phase: Implementation

Strategy = Libraries or Frameworks

Do not use dangerous functions such as gets. Look for their safe equivalent, which checks for the boundary.

Phase: Operation

Use OS-level preventative functionality. This is not a complete solution, but it provides some defense in depth.

Demonstrative Examples

Example 1:

While buffer overflow examples can be rather complex, it is possible to have very simple, yet still exploitable, heap-based buffer overflows:

Example Language: C *(Bad)*

```
#define BUFSIZE 256
int main(int argc, char **argv) {
  char *buf;
  buf = (char *)malloc(sizeof(char)*BUFSIZE);
  strcpy(buf, argv[1]);
}
```

The buffer is allocated heap memory with a fixed size, but there is no guarantee the string in argv[1] will not exceed this size and cause an overflow.

Example 2:

This example applies an encoding procedure to an input string and stores it into a buffer.

Example Language: C *(Bad)*

```
char * copy_input(char *user_supplied_string){
  int i, dst_index;
  char *dst_buf = (char*)malloc(4*sizeof(char) * MAX_SIZE);
  if ( MAX_SIZE <= strlen(user_supplied_string) ){
    die("user string too long, die evil hacker!");
  }
  dst_index = 0;
  for ( i = 0; i < strlen(user_supplied_string); i++ ){
    if( '&' == user_supplied_string[i] ){
      dst_buf[dst_index++] = '&';
      dst_buf[dst_index++] = 'a';
      dst_buf[dst_index++] = 'm';
      dst_buf[dst_index++] = 'p';
      dst_buf[dst_index++] = ';';
    }
    else if ('<' == user_supplied_string[i] ){
      /* encode to &lt; */
    }
    else dst_buf[dst_index++] = user_supplied_string[i];
  }
  return dst_buf;
}
```

The programmer attempts to encode the ampersand character in the user-controlled string, however the length of the string is validated before the encoding procedure is applied. Furthermore, the programmer assumes encoding expansion will only expand a given character by a factor of 4, while the encoding of the ampersand expands by 5. As a result, when the encoding procedure expands the string it is possible to overflow the destination buffer if the attacker provides a string of many ampersands.

Observed Examples

Reference	Description
CVE-2021-43537	Chain: in a web browser, an unsigned 64-bit integer is forcibly cast to a 32-bit integer (CWE-681) and potentially leading to an integer overflow (CWE-190). If an integer overflow occurs, this can cause heap memory corruption (CWE-122) *https://www.cve.org/CVERecord?id=CVE-2021-43537*
CVE-2007-4268	Chain: integer signedness error (CWE-195) passes signed comparison, leading to heap overflow (CWE-122) *https://www.cve.org/CVERecord?id=CVE-2007-4268*
CVE-2009-2523	Chain: product does not handle when an input string is not NULL terminated (CWE-170), leading to buffer over-read (CWE-125) or heap-based buffer overflow (CWE-122). *https://www.cve.org/CVERecord?id=CVE-2009-2523*
CVE-2021-29529	Chain: machine-learning product can have a heap-based buffer overflow (CWE-122) when some integer-oriented bounds are calculated by using ceiling() and floor() on floating point values (CWE-1339) *https://www.cve.org/CVERecord?id=CVE-2021-29529*
CVE-2010-1866	Chain: integer overflow (CWE-190) causes a negative signed value, which later bypasses a maximum-only check (CWE-839), leading to heap-based buffer overflow (CWE-122). *https://www.cve.org/CVERecord?id=CVE-2010-1866*

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Relationship

Heap-based buffer overflows are usually just as dangerous as stack-based buffer overflows.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Heap overflow
Software Fault Patterns	SFP8		Faulty Buffer Access
CERT C Secure Coding	STR31-C	CWE More Specific	Guarantee that storage for strings has sufficient space for character data and the null terminator
ISA/IEC 62443	Part 4-2		Req CR 3.5
ISA/IEC 62443	Part 3-3		Req SR 3.5
ISA/IEC 62443	Part 4-1		Req SI-1
ISA/IEC 62443	Part 4-1		Req SI-2
ISA/IEC 62443	Part 4-1		Req SVV-1
ISA/IEC 62443	Part 4-1		Req SVV-3

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
92	Forced Integer Overflow

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-58]Michael Howard. "Address Space Layout Randomization in Windows Vista". < https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista >.2023-04-07.

[REF-60]"PaX". < https://en.wikipedia.org/wiki/Executable_space_protection#PaX >.2023-04-07.

[REF-64]Grant Murphy. "Position Independent Executables (PIE)". 2012 November 8. Red Hat. < https://www.redhat.com/en/blog/position-independent-executables-pie >.2023-04-07.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-1337]Alexander Sotirov and Mark Dowd. "Bypassing Browser Memory Protections: Setting back browser security by 10 years". 2008. < https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf >.2023-04-26.

[REF-1332]John Richard Moser. "Prelink and address space randomization". 2006 July 5. < https://lwn.net/Articles/190139/ >.2023-04-26.

[REF-1333]Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". 2016. < http://www.cs.ucr.edu/~nael/pubs/micro16.pdf >.2023-04-26.

[REF-1334]D3FEND. "Stack Frame Canary Validation (D3-SFCV)". 2023. < https://d3fend.mitre.org/technique/d3f:StackFrameCanaryValidation/ >.2023-04-26.

[REF-1335]D3FEND. "Segment Address Offset Randomization (D3-SAOR)". 2023. < https://d3fend.mitre.org/technique/d3f:SegmentAddressOffsetRandomization/ >.2023-04-26.

CWE-123: Write-what-where Condition

Weakness ID : 123
Structure : Simple
Abstraction : Base

Description

Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	787	Out-of-bounds Write	1661
PeerOf	V	415	Double Free	1008
CanFollow	B	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
CanFollow	B	134	Use of Externally-Controlled Format String	365
CanFollow	B	364	Signal Handler Race Condition	899
CanFollow	V	416	Use After Free	1012
CanFollow	V	479	Signal Handler Use of a Non-reentrant Function	1147
CanFollow	V	590	Free of Memory not on the Heap	1326

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

Weakness Ordinalities

Resultant :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability Access Control	Modify Memory Execute Unauthorized Code or Commands Gain Privileges or Assume Identity DoS: Crash, Exit, or Restart Bypass Protection Mechanism *Clearly, write-what-where conditions can be used to write data to areas of memory outside the scope of a policy. Also, they almost invariably can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. If the attacker can overwrite a pointer's worth of memory (usually 32 or 64 bits), they can redirect a function pointer to their own malicious code. Even when the attacker can only modify a single byte arbitrary code execution can be possible. Sometimes this is because the same problem can be exploited repeatedly to the same effect. Other times it is because the attacker can overwrite security-critical application-specific data -- such as a flag indicating whether the user is an administrator.*	
Integrity	DoS: Crash, Exit, or Restart	

Scope	Impact	Likelihood
Availability	Modify Memory *Many memory accesses can lead to program termination, such as when writing to addresses that are invalid for the current process.*	
Access Control Other	Bypass Protection Mechanism Other *When the consequence is arbitrary code execution, this can often be used to subvert any other security service.*	

Potential Mitigations

Phase: Architecture and Design

Strategy = Language Selection

Use a language that provides appropriate memory abstractions.

Phase: Operation

Use OS-level preventative functionality integrated after the fact. Not a complete solution.

Demonstrative Examples

Example 1:

The classic example of a write-what-where condition occurs when the accounting information for memory allocations is overwritten in a particular fashion. Here is an example of potentially vulnerable code:

Example Language: C *(Bad)*

```
#define BUFSIZE 256
int main(int argc, char **argv) {
  char *buf1 = (char *) malloc(BUFSIZE);
  char *buf2 = (char *) malloc(BUFSIZE);
  strcpy(buf1, argv[1]);
  free(buf2);
}
```

Vulnerability in this case is dependent on memory layout. The call to strcpy() can be used to write past the end of buf1, and, with a typical layout, can overwrite the accounting information that the system keeps for buf2 when it is allocated. Note that if the allocation header for buf2 can be overwritten, buf2 itself can be overwritten as well.

The allocation header will generally keep a linked list of memory "chunks". Particularly, there may be a "previous" chunk and a "next" chunk. Here, the previous chunk for buf2 will probably be buf1, and the next chunk may be null. When the free() occurs, most memory allocators will rewrite the linked list using data from buf2. Particularly, the "next" chunk for buf1 will be updated and the "previous" chunk for any subsequent chunk will be updated. The attacker can insert a memory address for the "next" chunk and a value to write into that memory address for the "previous" chunk.

This could be used to overwrite a function pointer that gets dereferenced later, replacing it with a memory address that the attacker has legitimate access to, where they have placed malicious code, resulting in arbitrary code execution.

Observed Examples

Reference	Description
CVE-2022-21668	Chain: Python library does not limit the resources used to process images that specify a very large number of bands (CWE-1284), leading to excessive memory consumption (CWE-789) or an integer overflow (CWE-190).

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2022-21668
CVE-2022-0545	Chain: 3D renderer has an integer overflow (CWE-190) leading to write-what-where condition (CWE-123) using a crafted image. *https://www.cve.org/CVERecord?id=CVE-2022-0545*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Write-what-where condition
CERT C Secure Coding	ARR30-C	Imprecise	Do not form or use out-of-bounds pointers or array subscripts
CERT C Secure Coding	ARR38-C	Imprecise	Guarantee that library functions do not form invalid pointers
CERT C Secure Coding	STR31-C	Imprecise	Guarantee that storage for strings has sufficient space for character data and the null terminator
CERT C Secure Coding	STR32-C	Imprecise	Do not pass a non-null-terminated character sequence to a library function that expects a string
Software Fault Patterns	SFP8		Faulty Buffer Access

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-124: Buffer Underwrite ('Buffer Underflow')

Weakness ID : 124
Structure : Simple
Abstraction : Base

Description

The product writes to a buffer using an index or pointer that references a memory location prior to the beginning of the buffer.

Extended Description

This typically occurs when a pointer or its index is decremented to a position before the buffer, when pointer arithmetic results in a position before the beginning of the valid memory location, or when a negative index is used.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	787	Out-of-bounds Write	1661
ChildOf	B	786	Access of Memory Location Before Start of Buffer	1658
CanFollow	B	839	Numeric Range Comparison Without Minimum Check	1767

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

Nature	Type	ID	Name	Page
MemberOf	C	1218	Memory Buffer Errors	2479

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Alternate Terms

buffer underrun : Some prominent vendors and researchers use the term "buffer underrun". "Buffer underflow" is more commonly used, although both terms are also sometimes used to describe a buffer under-read (CWE-127).

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Integrity Availability	Modify Memory DoS: Crash, Exit, or Restart *Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash.*	
Integrity Confidentiality Availability Access Control Other	Execute Unauthorized Code or Commands Modify Memory Bypass Protection Mechanism Other *If the corrupted memory can be effectively controlled, it may be possible to execute arbitrary code. If the corrupted memory is data rather than instructions, the system will continue to function with improper changes, possibly in violation of an implicit or explicit policy. The consequences would only be limited by how the affected data is used, such as an adjacent memory location that is used to specify whether the user has special privileges.*	
Access Control Other	Bypass Protection Mechanism Other	

Scope	Impact	Likelihood
	When the consequence is arbitrary code execution, this can often be used to subvert any other security service.	

Potential Mitigations

Phase: Requirements

Choose a language that is not susceptible to these issues.

Phase: Implementation

All calculated values that are used as index or for pointer arithmetic should be validated to ensure that they are within an expected range.

Demonstrative Examples

Example 1:

In the following C/C++ example, a utility function is used to trim trailing whitespace from a character string. The function copies the input string to a local character string and uses a while statement to remove the trailing whitespace by moving backward through the string and overwriting whitespace with a NUL character.

Example Language: C *(Bad)*

```
char* trimTrailingWhitespace(char *strMessage, int length) {
  char *retMessage;
  char *message = malloc(sizeof(char)*(length+1));
  // copy input string to a temporary string
  char message[length+1];
  int index;
  for (index = 0; index < length; index++) {
    message[index] = strMessage[index];
  }
  message[index] = '\0';
  // trim trailing whitespace
  int len = index-1;
  while (isspace(message[len])) {
    message[len] = '\0';
    len--;
  }
  // return string without trailing whitespace
  retMessage = message;
  return retMessage;
}
```

However, this function can cause a buffer underwrite if the input character string contains all whitespace. On some systems the while statement will move backwards past the beginning of a character string and will call the isspace() function on an address outside of the bounds of the local buffer.

Example 2:

The following is an example of code that may result in a buffer underwrite. This code is attempting to replace the substring "Replace Me" in destBuf with the string stored in srcBuf. It does so by using the function strstr(), which returns a pointer to the found substring in destBuf. Using pointer arithmetic, the starting index of the substring is found.

Example Language: C *(Bad)*

```
int main() {
  ...
  char *result = strstr(destBuf, "Replace Me");
  int idx = result - destBuf;
  strcpy(&destBuf[idx], srcBuf);
  ...
}
```

In the case where the substring is not found in destBuf, strstr() will return NULL, causing the pointer arithmetic to be undefined, potentially setting the value of idx to a negative number. If idx is negative, this will result in a buffer underwrite of destBuf.

Observed Examples

Reference	Description
CVE-2021-24018	buffer underwrite in firmware verification routine allows code execution via a crafted firmware image *https://www.cve.org/CVERecord?id=CVE-2021-24018*
CVE-2002-2227	Unchecked length of SSLv2 challenge value leads to buffer underflow. *https://www.cve.org/CVERecord?id=CVE-2002-2227*
CVE-2007-4580	Buffer underflow from a small size value with a large buffer (length parameter inconsistency, CWE-130) *https://www.cve.org/CVERecord?id=CVE-2007-4580*
CVE-2007-1584	Buffer underflow from an all-whitespace string, which causes a counter to be decremented before the buffer while looking for a non-whitespace character. *https://www.cve.org/CVERecord?id=CVE-2007-1584*
CVE-2007-0886	Buffer underflow resultant from encoded data that triggers an integer overflow. *https://www.cve.org/CVERecord?id=CVE-2007-0886*
CVE-2006-6171	Product sets an incorrect buffer size limit, leading to "off-by-two" buffer underflow. *https://www.cve.org/CVERecord?id=CVE-2006-6171*
CVE-2006-4024	Negative value is used in a memcpy() operation, leading to buffer underflow. *https://www.cve.org/CVERecord?id=CVE-2006-4024*
CVE-2004-2620	Buffer underflow due to mishandled special characters *https://www.cve.org/CVERecord?id=CVE-2004-2620*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Relationship

This could be resultant from several errors, including a bad offset or an array index that decrements before the beginning of the buffer (see CWE-129).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			UNDER - Boundary beginning violation ('buffer underflow'?)
CLASP			Buffer underwrite
Software Fault Patterns	SFP8		Faulty Buffer Access

References

[REF-90]"Buffer UNDERFLOWS: What do you know about it?". Vuln-Dev Mailing List. 2004 January 0. < https://seclists.org/vuln-dev/2004/Jan/22 >.2023-04-07.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-125: Out-of-bounds Read

Weakness ID : 125
Structure : Simple
Abstraction : Base

Description

The product reads data past the end, or before the beginning, of the intended buffer.

Extended Description

Typically, this can allow attackers to read sensitive information from other memory locations or cause a crash. A crash can occur when the code reads a variable amount of data and assumes that a sentinel exists to stop the read operation, such as a NUL in a string. The expected sentinel might not be located in the out-of-bounds memory, causing excessive data to be read, leading to a segmentation fault or a buffer overflow. The product may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent read operation then produces undefined or unexpected results.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293
ParentOf	V	126	Buffer Over-read	334
ParentOf	V	127	Buffer Under-read	337
CanFollow	B	822	Untrusted Pointer Dereference	1723
CanFollow	B	823	Use of Out-of-range Pointer Offset	1726
CanFollow	B	824	Access of Uninitialized Pointer	1729
CanFollow	B	825	Expired Pointer Dereference	1732

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
MemberOf	C	1218	Memory Buffer Errors	2479

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Technology : ICS/OT *(Prevalence = Often)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Memory	
Confidentiality	Bypass Protection Mechanism *By reading out-of-bounds memory, an attacker might be able to get secret values, such as memory addresses, which can be bypass protection mechanisms such as ASLR in order to improve the reliability and likelihood of exploiting a separate weakness to achieve code execution instead of just denial of service.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

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

Potential Mitigations

Phase: 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. To reduce the likelihood of introducing an out-of-bounds read, ensure that you validate and ensure correct calculations for any length argument, buffer size calculation, or offset. Be especially careful of relying on a sentinel (i.e. special character such as NUL) in untrusted inputs.

Phase: Architecture and Design

Strategy = Language Selection

Use a language that provides appropriate memory abstractions.

Demonstrative Examples

Example 1:

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method

Example Language: C *(Bad)*

```
int getValueFromArray(int *array, int len, int index) {
  int value;
  // check that the array index is less than the maximum
  // length of the array
  if (index < len) {
    // get the value at the specified index of the array
    value = array[index];
  }
  // if array index is invalid then output error message
  // and return value indicating error
  else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }
  return value;
}
```

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

Example Language: C *(Good)*

```
...
// check that the array index is within the correct
// range of values for the array
if (index >= 0 && index < len) {
...
```

Observed Examples

Reference	Description
CVE-2020-11899	Out-of-bounds read in IP stack used in embedded systems, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2020-11899*
CVE-2014-0160	Chain: "Heartbleed" bug receives an inconsistent length parameter (CWE-130) enabling an out-of-bounds read (CWE-126), returning memory that could include private cryptographic keys and other sensitive data. *https://www.cve.org/CVERecord?id=CVE-2014-0160*

Reference	Description
CVE-2021-40985	HTML conversion package has a buffer under-read, allowing a crash *https://www.cve.org/CVERecord?id=CVE-2021-40985*
CVE-2018-10887	Chain: unexpected sign extension (CWE-194) leads to integer overflow (CWE-190), causing an out-of-bounds read (CWE-125) *https://www.cve.org/CVERecord?id=CVE-2018-10887*
CVE-2009-2523	Chain: product does not handle when an input string is not NULL terminated (CWE-170), leading to buffer over-read (CWE-125) or heap-based buffer overflow (CWE-122). *https://www.cve.org/CVERecord?id=CVE-2009-2523*
CVE-2018-16069	Chain: series of floating-point precision errors (CWE-1339) in a web browser rendering engine causes out-of-bounds read (CWE-125), giving access to cross-origin data *https://www.cve.org/CVERecord?id=CVE-2018-16069*
CVE-2004-0112	out-of-bounds read due to improper length check *https://www.cve.org/CVERecord?id=CVE-2004-0112*
CVE-2004-0183	packet with large number of specified elements cause out-of-bounds read. *https://www.cve.org/CVERecord?id=CVE-2004-0183*
CVE-2004-0221	packet with large number of specified elements cause out-of-bounds read. *https://www.cve.org/CVERecord?id=CVE-2004-0221*
CVE-2004-0184	out-of-bounds read, resultant from integer underflow *https://www.cve.org/CVERecord?id=CVE-2004-0184*
CVE-2004-1940	large length value causes out-of-bounds read *https://www.cve.org/CVERecord?id=CVE-2004-1940*
CVE-2004-0421	malformed image causes out-of-bounds read *https://www.cve.org/CVERecord?id=CVE-2004-0421*
CVE-2008-4113	OS kernel trusts userland-supplied length value, allowing reading of sensitive information *https://www.cve.org/CVERecord?id=CVE-2008-4113*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1157	SEI CERT C Coding Standard - Guidelines 03. Expressions (EXP)	1154	2455
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	C	1366	ICS Communications: Frail Security in Protocols	1358	2503
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	1387	2597
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Nature	Type	ID	Name	V	Page
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses	1425	2600

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Out-of-bounds Read
CERT C Secure Coding	ARR30-C	Imprecise	Do not form or use out-of-bounds pointers or array subscripts
CERT C Secure Coding	ARR38-C	Imprecise	Guarantee that library functions do not form invalid pointers
CERT C Secure Coding	EXP39-C	Imprecise	Do not access a variable through a pointer of an incompatible type
CERT C Secure Coding	STR31-C	Imprecise	Guarantee that storage for strings has sufficient space for character data and the null terminator
CERT C Secure Coding	STR32-C	CWE More Abstract	Do not pass a non-null-terminated character sequence to a library function that expects a string
Software Fault Patterns	SFP8		Faulty Buffer Access

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
540	Overread Buffers

References

[REF-1034]Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund and Thomas Walter. "Breaking the memory secrecy assumption". 2009 March 1. ACM. < https://dl.acm.org/doi/10.1145/1519144.1519145 >.2023-04-07.

[REF-1035]Fermin J. Serna. "The info leak era on software exploitation". 2012 July 5. < https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-126: Buffer Over-read

Weakness ID : 126
Structure : Simple
Abstraction : Variant

Description

The product reads from a buffer using buffer access mechanisms such as indexes or pointers that reference memory locations after the targeted buffer.

Extended Description

This typically occurs when the pointer or its index is incremented to a position beyond the bounds of the buffer or when pointer arithmetic results in a position outside of the valid memory location to name a few. This may result in exposure of sensitive information or possibly a crash.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	125	Out-of-bounds Read	330
ChildOf	B	788	Access of Memory Location After End of Buffer	1669
CanFollow	B	170	Improper Null Termination	428

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Memory	
Confidentiality	Bypass Protection Mechanism *By reading out-of-bounds memory, an attacker might be able to get secret values, such as memory addresses, which can be bypass protection mechanisms such as ASLR in order to improve the reliability and likelihood of exploiting a separate weakness to achieve code execution instead of just denial of service.*	

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

Demonstrative Examples

Example 1:

In the following C/C++ example the method processMessageFromSocket() will get a message from a socket, placed into a buffer, and will parse the contents of the buffer into a structure that contains the message length and the message body. A for loop is used to copy the message body into a local character string which will be passed to another method for processing.

Example Language: C *(Bad)*

```
int processMessageFromSocket(int socket) {
  int success;
  char buffer[BUFFER_SIZE];
  char message[MESSAGE_SIZE];
  // get message from socket and store into buffer
  //Ignoring possibliity that buffer > BUFFER_SIZE
  if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
    // place contents of the buffer into message structure
    ExMessage *msg = recastBuffer(buffer);
    // copy message body into string for processing
    int index;
    for (index = 0; index < msg->msgLength; index++) {
      message[index] = msg->msgBody[index];
```

```
        }
        message[index] = '\0';
        // process message
        success = processMessage(message);
    }
    return success;
}
```

However, the message length variable from the structure is used as the condition for ending the for loop without validating that the message length variable accurately reflects the length of the message body (CWE-606). This can result in a buffer over-read (CWE-125) by reading from memory beyond the bounds of the buffer if the message length variable indicates a length that is longer than the size of a message body (CWE-130).

Example 2:

The following C/C++ example demonstrates a buffer over-read due to a missing NULL terminator. The main method of a pattern matching utility that looks for a specific pattern within a specific file uses the string strncopy() method to copy the command line user input file name and pattern to the Filename and Pattern character arrays respectively.

Example Language: C *(Bad)*

```
int main(int argc, char **argv)
{
    char Filename[256];
    char Pattern[32];
    /* Validate number of parameters and ensure valid content */
    ...
    /* copy filename parameter to variable, may cause off-by-one overflow */
    strncpy(Filename, argv[1], sizeof(Filename));
    /* copy pattern parameter to variable, may cause off-by-one overflow */
    strncpy(Pattern, argv[2], sizeof(Pattern));
    printf("Searching file: %s for the pattern: %s\n", Filename, Pattern);
    Scan_File(Filename, Pattern);
}
```

However, the code do not take into account that strncpy() will not add a NULL terminator when the source buffer is equal in length of longer than that provide size attribute. Therefore if a user enters a filename or pattern that are the same size as (or larger than) their respective character arrays, a NULL terminator will not be added (CWE-170) which leads to the printf() read beyond the expected end of the Filename and Pattern buffers.

To fix this problem, be sure to subtract 1 from the sizeof() call to allow room for the null byte to be added.

Example Language: C *(Good)*

```
/* copy filename parameter to variable, no off-by-one overflow */
strncpy(Filename, argv[2], sizeof(Filename)-1);
Filename[255]='\0';
/* copy pattern parameter to variable, no off-by-one overflow */
strncpy(Pattern, argv[3], sizeof(Pattern)-1);
Pattern[31]='\0';
```

Observed Examples

Reference	Description
CVE-2022-1733	Text editor has out-of-bounds read past end of line while indenting C code *https://www.cve.org/CVERecord?id=CVE-2022-1733*
CVE-2014-0160	Chain: "Heartbleed" bug receives an inconsistent length parameter (CWE-130) enabling an out-of-bounds read (CWE-126), returning memory that could include private cryptographic keys and other sensitive data.

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2014-0160
CVE-2009-2523	Chain: product does not handle when an input string is not NULL terminated, leading to buffer over-read or heap-based buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2009-2523*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Relationship

These problems may be resultant from missing sentinel values (CWE-463) or trusting a user-influenced input length variable.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Buffer over-read
Software Fault Patterns	SFP8		Faulty Buffer Access

References

[REF-1034]Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund and Thomas Walter. "Breaking the memory secrecy assumption". 2009 March 1. ACM. < https://dl.acm.org/doi/10.1145/1519144.1519145 >.2023-04-07.

[REF-1035]Fermin J. Serna. "The info leak era on software exploitation". 2012 July 5. < https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-127: Buffer Under-read

Weakness ID : 127
Structure : Simple
Abstraction : Variant

Description

The product reads from a buffer using buffer access mechanisms such as indexes or pointers that reference memory locations prior to the targeted buffer.

Extended Description

This typically occurs when the pointer or its index is decremented to a position before the buffer, when pointer arithmetic results in a position before the beginning of the valid memory location, or when a negative index is used. This may result in exposure of sensitive information or possibly a crash.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	125	Out-of-bounds Read	330
ChildOf	B	786	Access of Memory Location Before Start of Buffer	1658

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Memory	
Confidentiality	Bypass Protection Mechanism *By reading out-of-bounds memory, an attacker might be able to get secret values, such as memory addresses, which can be bypass protection mechanisms such as ASLR in order to improve the reliability and likelihood of exploiting a separate weakness to achieve code execution instead of just denial of service.*	

Observed Examples

Reference	Description
CVE-2021-40985	HTML conversion package has a buffer under-read, allowing a crash *https://www.cve.org/CVERecord?id=CVE-2021-40985*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Research Gap

Under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Buffer under-read
Software Fault Patterns	SFP8		Faulty Buffer Access

References

[REF-1034]Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund and Thomas Walter. "Breaking the memory secrecy assumption". 2009 March 1. ACM. < https://dl.acm.org/doi/10.1145/1519144.1519145 >.2023-04-07.

[REF-1035]Fermin J. Serna. "The info leak era on software exploitation". 2012 July 5. < https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-128: Wrap-around Error

Weakness ID : 128
Structure : Simple
Abstraction : Base

Description

Wrap around errors occur whenever a value is incremented past the maximum value for its type and therefore "wraps around" to a very small, negative, or undefined value.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499
PeerOf	B	190	Integer Overflow or Wraparound	472
CanPrecede	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
MemberOf	C	189	Numeric Errors	2312

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Often)*

Language : C++ *(Prevalence = Often)*

Background Details

Due to how addition is performed by computers, if a primitive is incremented past the maximum value possible for its storage space, the system will not recognize this, and therefore increment each bit as if it still had extra space. Because of how negative numbers are represented in binary, primitives interpreted as signed may "wrap" to very large negative values.

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) DoS: Instability	

Scope	Impact	Likelihood
	This weakness will generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.	
Integrity	Modify Memory *If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the wrap around results in other conditions such as buffer overflows, further memory corruption may occur.*	
Confidentiality Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism *This weakness can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.*	

Potential Mitigations

Requirements specification: The choice could be made to use a language that is not susceptible to these issues.

Phase: Architecture and Design

Provide clear upper and lower bounds on the scale of any protocols designed.

Phase: Implementation

Perform validation on all incremented variables to ensure that they remain within reasonable bounds.

Demonstrative Examples

Example 1:

The following image processing code allocates a table for images.

Example Language: C *(Bad)*

```
img_t table_ptr; /*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

This code intends to allocate a table of size num_imgs, however as num_imgs grows large, the calculation determining the size of the list will eventually overflow (CWE-190). This will result in a very small list to be allocated instead. If the subsequent code operates on the list as if it were num_imgs long, it may result in many types of out-of-bounds problems (CWE-119).

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Notes

Relationship

The relationship between overflow and wrap-around needs to be examined more closely, since several entries (including CWE-190) are closely related.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Wrap-around error
CERT C Secure Coding	MEM07-C		Ensure that the arguments to calloc(), when multiplied, can be represented as a size_t
Software Fault Patterns	SFP1		Glitch in computation

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
92	Forced Integer Overflow

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-129: Improper Validation of Array Index

Weakness ID : 129
Structure : Simple
Abstraction : Variant

Description

The product uses untrusted input when calculating or using an array index, but the product does not validate or incorrectly validates the index to ensure the index references a valid position within the array.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	1285	Improper Validation of Specified Index, Position, or Offset in Input	2132
CanPrecede	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293
CanPrecede	V	789	Memory Allocation with Excessive Size Value	1674

Nature	Type	ID	Name	Page
CanPrecede	B	823	Use of Out-of-range Pointer Offset	1726

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Resultant : The most common condition situation leading to an out-of-bounds array index is the use of loop index variables as buffer indexes. If the end condition for the loop is subject to a flaw, the index can grow or shrink unbounded, therefore causing a buffer overflow or underflow. Another common situation leading to this condition is the use of a function's return value, or the resulting value of a calculation directly as an index in to a buffer.

Applicable Platforms

Language : C *(Prevalence = Often)*

Language : C++ *(Prevalence = Often)*

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

Alternate Terms

out-of-bounds array index :

index-out-of-range :

array index underflow :

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Availability	DoS: Crash, Exit, or Restart *Use of an index that is outside the bounds of an array will very likely result in the corruption of relevant memory and perhaps instructions, leading to a crash, if the values are outside of the valid memory area.*	
Integrity	Modify Memory *If the memory corrupted is data, rather than instructions, the system will continue to function with improper values.*	
Confidentiality Integrity	Modify Memory Read Memory *Use of an index that is outside the bounds of an array can also trigger out-of-bounds read or write operations, or operations on the wrong objects; i.e., "buffer overflows" are not always the result. This may result in the exposure or modification of sensitive data.*	
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands *If the memory accessible by the attacker can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow and possibly without the use of large inputs if a precise index can be controlled.*	
Integrity Availability	DoS: Crash, Exit, or Restart Execute Unauthorized Code or Commands	

Scope	Impact	Likelihood
Confidentiality	Read Memory Modify Memory *A single fault could allow either an overflow (CWE-788) or underflow (CWE-786) of the array index. What happens next will depend on the type of operation being performed out of bounds, but can expose sensitive information, cause a system crash, or possibly lead to arbitrary code execution.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives. Automated static analysis generally does not account for environmental considerations when reporting out-of-bounds memory operations. This can make it difficult for users to determine which warnings should be investigated first. For example, an analysis tool might report array index errors that originate from command line arguments in a program that is not expected to run with setuid or other special privileges.

Effectiveness = High

This is not a perfect solution, since 100% accuracy and coverage are not feasible.

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Black Box

Black box methods might not get the needed code coverage within limited time constraints, and a dynamic test might not produce any noticeable side effects even if it is successful.

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Use an input validation framework such as Struts or the OWASP ESAPI Validation API. Note that using a framework does not automatically address all input validation problems; be mindful of weaknesses that could arise from misusing the framework itself (CWE-1173).

Phase: 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. Even though client-side checks provide minimal benefits with respect to server-side security, they are still useful. First, they can support intrusion detection. If the server receives input that should have been rejected by the client, then it may be an indication of an attack. Second, client-side error-checking can provide helpful feedback to the user about the expectations for valid input. Third, there may be a reduction in server-side processing time for accidental input errors, although this is typically a small savings.

Phase: Requirements

Strategy = Language Selection

Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, Ada allows the programmer to constrain the values of a variable and languages such as Java and Ruby will allow the programmer to handle exceptions when an out-of-bounds index is accessed.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335].

Effectiveness = Defense in Depth

These techniques do not provide a complete solution. For instance, exploits frequently use a bug that discloses memory addresses in order to maximize reliability of code execution [REF-1337]. It has also been shown that a side-channel attack can bypass ASLR [REF-1333]

Phase: Operation

Strategy = Environment Hardening

Use a CPU and operating system that offers Data Execution Protection (using hardware NX or XD bits) or the equivalent techniques that simulate this feature in software, such as PaX [REF-60] [REF-61]. These techniques ensure that any instruction executed is exclusively at a memory address that is part of the code segment. For more information on these techniques see D3-PSEP (Process Segment Execution Prevention) from D3FEND [REF-1336].

Effectiveness = Defense in Depth

This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Phase: 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 accessing a user-controlled array index, use a stringent range of values that are within the target array. Make sure that you do not allow negative values to be used. That is, verify the minimum as well as the maximum of the range of acceptable values.

Phase: Implementation

Be especially careful to validate all input when invoking code that crosses language boundaries, such as from an interpreted language to native code. This could create an unexpected interaction between the language boundaries. Ensure that you are not violating any of the expectations of the language with which you are interfacing. For example, even though Java may not be susceptible to buffer overflows, providing a large argument in a call to native code might trigger an overflow.

Phase: Architecture and Design

Phase: Operation

Strategy = Environment Hardening

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.

Phase: Architecture and Design

Phase: Operation

Strategy = Sandbox or Jail

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

The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Demonstrative Examples

Example 1:

In the code snippet below, an untrusted integer value is used to reference an object in an array.

Example Language: Java *(Bad)*

```
public String getValue(int index) {
  return array[index];
}
```

If index is outside of the range of the array, this may result in an ArrayIndexOutOfBounds Exception being raised.

Example 2:

The following example takes a user-supplied value to allocate an array of objects and then operates on the array.

Example Language: Java *(Bad)*

```
private void buildList ( int untrustedListSize ){
  if ( 0 > untrustedListSize ){
    die("Negative value supplied for list size, die evil hacker!");
  }
  Widget[] list = new Widget [ untrustedListSize ];
  list[0] = new Widget();
}
```

This example attempts to build a list from a user-specified value, and even checks to ensure a non-negative value is supplied. If, however, a 0 value is provided, the code will build an array of size 0 and then try to store a new Widget in the first location, causing an exception to be thrown.

Example 3:

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method

Example Language: C *(Bad)*

```
int getValueFromArray(int *array, int len, int index) {
  int value;
  // check that the array index is less than the maximum
  // length of the array
  if (index < len) {
    // get the value at the specified index of the array
    value = array[index];
  }
  // if array index is invalid then output error message
  // and return value indicating error
  else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }
  return value;
}
```

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

Example Language: C *(Good)*

```
...
// check that the array index is within the correct
// range of values for the array
if (index >= 0 && index < len) {
...
```

Example 4:

The following example retrieves the sizes of messages for a pop3 mail server. The message sizes are retrieved from a socket that returns in a buffer the message number and the message size, the message number (num) and size (size) are extracted from the buffer and the message size is placed into an array using the message number for the array index.

Example Language: C *(Bad)*

```
/* capture the sizes of all messages */
int getsizes(int sock, int count, int *sizes) {
    ...
    char buf[BUFFER_SIZE];
    int ok;
    int num, size;
    // read values from socket and added to sizes array
    while ((ok = gen_recv(sock, buf, sizeof(buf))) == 0)
    {
        // continue read from socket until buf only contains '.'
        if (DOTLINE(buf))
            break;
        else if (sscanf(buf, "%d %d", &num, &size) == 2)
            sizes[num - 1] = size;
    }
        ...
}
```

In this example the message number retrieved from the buffer could be a value that is outside the allowable range of indices for the array and could possibly be a negative number. Without proper validation of the value to be used for the array index an array overflow could occur and could potentially lead to unauthorized access to memory addresses and system crashes. The value of the array index should be validated to ensure that it is within the allowable range of indices for the array as in the following code.

Example Language: C *(Good)*

```
/* capture the sizes of all messages */
int getsizes(int sock, int count, int *sizes) {
    ...
    char buf[BUFFER_SIZE];
    int ok;
    int num, size;
    // read values from socket and added to sizes array
    while ((ok = gen_recv(sock, buf, sizeof(buf))) == 0)
    {
        // continue read from socket until buf only contains '.'
        if (DOTLINE(buf))
            break;
        else if (sscanf(buf, "%d %d", &num, &size) == 2) {
            if (num > 0 && num <= (unsigned)count)
                sizes[num - 1] = size;
            else
                /* warn about possible attempt to induce buffer overflow */
                report(stderr, "Warning: ignoring bogus data for message sizes returned by server.\n");
        }
    }
        ...
}
```

Example 5:

In the following example the method displayProductSummary is called from a Web service servlet to retrieve product summary information for display to the user. The servlet obtains the integer value of the product number from the user and passes it to the displayProductSummary method. The displayProductSummary method passes the integer value of the product number to the getProductSummary method which obtains the product summary from the array object containing the project summaries using the integer value of the product number as the array index.

Example Language: Java *(Bad)*

```
// Method called from servlet to obtain product information
public String displayProductSummary(int index) {
    String productSummary = new String("");
    try {
        String productSummary = getProductSummary(index);
    } catch (Exception ex) {...}
    return productSummary;
}
public String getProductSummary(int index) {
    return products[index];
}
```

In this example the integer value used as the array index that is provided by the user may be outside the allowable range of indices for the array which may provide unexpected results or cause the application to fail. The integer value used for the array index should be validated to ensure that it is within the allowable range of indices for the array as in the following code.

Example Language: Java *(Good)*

```
// Method called from servlet to obtain product information
public String displayProductSummary(int index) {
    String productSummary = new String("");
    try {
        String productSummary = getProductSummary(index);
    } catch (Exception ex) {...}
    return productSummary;
}
public String getProductSummary(int index) {
    String productSummary = "";
    if ((index >= 0) && (index < MAX_PRODUCTS)) {
        productSummary = products[index];
    }
    else {
        System.err.println("index is out of bounds");
        throw new IndexOutOfBoundsException();
    }
    return productSummary;
}
```

An alternative in Java would be to use one of the collection objects such as ArrayList that will automatically generate an exception if an attempt is made to access an array index that is out of bounds.

Example Language: Java *(Good)*

```
ArrayList productArray = new ArrayList(MAX_PRODUCTS);
...
try {
    productSummary = (String) productArray.get(index);
} catch (IndexOutOfBoundsException ex) {...}
```

Example 6:

The following example asks a user for an offset into an array to select an item.

Example Language: C *(Bad)*

```
int main (int argc, char **argv) {
    char *items[] = {"boat", "car", "truck", "train"};
    int index = GetUntrustedOffset();
    printf("You selected %s\n", items[index-1]);
}
```

The programmer allows the user to specify which element in the list to select, however an attacker can provide an out-of-bounds offset, resulting in a buffer over-read (CWE-126).

Observed Examples

Reference	Description
CVE-2005-0369	large ID in packet used as array index

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2005-0369
CVE-2001-1009	negative array index as argument to POP LIST command *https://www.cve.org/CVERecord?id=CVE-2001-1009*
CVE-2003-0721	Integer signedness error leads to negative array index *https://www.cve.org/CVERecord?id=CVE-2003-0721*
CVE-2004-1189	product does not properly track a count and a maximum number, which can lead to resultant array index overflow. *https://www.cve.org/CVERecord?id=CVE-2004-1189*
CVE-2007-5756	Chain: device driver for packet-capturing software allows access to an unintended IOCTL with resultant array index error. *https://www.cve.org/CVERecord?id=CVE-2007-5756*
CVE-2005-2456	Chain: array index error (CWE-129) leads to deadlock (CWE-833) *https://www.cve.org/CVERecord?id=CVE-2005-2456*

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	740	CERT C Secure Coding Standard (2008) Chapter 7 - Arrays (ARR)	734	2344
MemberOf	C	802	2010 Top 25 - Risky Resource Management	800	2354
MemberOf	C	867	2011 Top 25 - Weaknesses On the Cusp	900	2372
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	C	874	CERT C++ Secure Coding Section 06 - Arrays and the STL (ARR)	868	2375
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	970	SFP Secondary Cluster: Faulty Buffer Access	888	2405
MemberOf	C	1131	CISQ Quality Measures (2016) - Security	1128	2442
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1179	SEI CERT Perl Coding Standard - Guidelines 01. Input Validation and Data Sanitization (IDS)	1178	2465
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Relationship

This weakness can precede uncontrolled memory allocation (CWE-789) in languages that automatically expand an array when an index is used that is larger than the size of the array, such as JavaScript.

Theoretical

An improperly validated array index might lead directly to the always-incorrect behavior of "access of array using out-of-bounds index."

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Unchecked array indexing
PLOVER			INDEX - Array index overflow
CERT C Secure Coding	ARR00-C		Understand how arrays work
CERT C Secure Coding	ARR30-C	CWE More Specific	Do not form or use out-of-bounds pointers or array subscripts
CERT C Secure Coding	ARR38-C		Do not add or subtract an integer to a pointer if the resulting value does not refer to a valid array element
CERT C Secure Coding	INT32-C		Ensure that operations on signed integers do not result in overflow
SEI CERT Perl Coding Standard	IDS32-PL	Imprecise	Validate any integer that is used as an array index
OMG ASCSM	ASCSM-CWE-129		
Software Fault Patterns	SFP8		Faulty Buffer Access

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
100	Overflow Buffers

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-96]Jason Lam. "Top 25 Series - Rank 14 - Improper Validation of Array Index". 2010 March 2. SANS Software Security Institute. < https://web.archive.org/web/20100316064026/http://blogs.sans.org/appsecstreetfighter/2010/03/12/top-25-series-rank-14-improper-validation-of-array-index/ >.2023-04-07.

[REF-58]Michael Howard. "Address Space Layout Randomization in Windows Vista". < https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista >.2023-04-07.

[REF-60]"PaX". < https://en.wikipedia.org/wiki/Executable_space_protection#PaX >.2023-04-07.

[REF-61]Microsoft. "Understanding DEP as a mitigation technology part 1". < https://msrc.microsoft.com/blog/2009/06/understanding-dep-as-a-mitigation-technology-part-1/ >.2023-04-07.

[REF-76]Sean Barnum and Michael Gegick. "Least Privilege". 2005 September 4. < https://web.archive.org/web/20211209014121/https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege >.2023-04-07.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-64]Grant Murphy. "Position Independent Executables (PIE)". 2012 November 8. Red Hat. < https://www.redhat.com/en/blog/position-independent-executables-pie >.2023-04-07.

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

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-1332]John Richard Moser. "Prelink and address space randomization". 2006 July 5. < https://lwn.net/Articles/190139/ >.2023-04-26.

[REF-1333]Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". 2016. < http://www.cs.ucr.edu/~nael/pubs/micro16.pdf >.2023-04-26.

[REF-1335]D3FEND. "Segment Address Offset Randomization (D3-SAOR)". 2023. < https://d3fend.mitre.org/technique/d3f:SegmentAddressOffsetRandomization/ >.2023-04-26.

[REF-1336]D3FEND. "Process Segment Execution Prevention (D3-PSEP)". 2023. < https://d3fend.mitre.org/technique/d3f:ProcessSegmentExecutionPrevention/ >.2023-04-26.

[REF-1337]Alexander Sotirov and Mark Dowd. "Bypassing Browser Memory Protections: Setting back browser security by 10 years". 2008. < https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf >.2023-04-26.

CWE-130: Improper Handling of Length Parameter Inconsistency

Weakness ID : 130
Structure : Simple
Abstraction : Base

Description

The product parses a formatted message or structure, but it does not handle or incorrectly handles a length field that is inconsistent with the actual length of the associated data.

Extended Description

If an attacker can manipulate the length parameter associated with an input such that it is inconsistent with the actual length of the input, this can be leveraged to cause the target application to behave in unexpected, and possibly, malicious ways. One of the possible motives for doing so is to pass in arbitrarily large input to the application. Another possible motivation is the modification of application state by including invalid data for subsequent properties of the application. Such weaknesses commonly lead to attacks such as buffer overflows and execution of arbitrary code.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	240	Improper Handling of Inconsistent Structural Elements	583
CanPrecede	Ⓑ	805	Buffer Access with Incorrect Length Value	1702

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
MemberOf	🄲	19	Data Processing Errors	2309

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Sometimes)*

Language : C++ *(Prevalence = Sometimes)*

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

Alternate Terms

length manipulation :

length tampering :

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Memory Modify Memory Varies by Context	

Potential Mitigations

Phase: Implementation

When processing structured incoming data containing a size field followed by raw data, ensure that you identify and resolve any inconsistencies between the size field and the actual size of the data.

Phase: Implementation

Do not let the user control the size of the buffer.

Phase: Implementation

Validate that the length of the user-supplied data is consistent with the buffer size.

Demonstrative Examples

Example 1:

In the following C/C++ example the method processMessageFromSocket() will get a message from a socket, placed into a buffer, and will parse the contents of the buffer into a structure that contains the message length and the message body. A for loop is used to copy the message body into a local character string which will be passed to another method for processing.

Example Language: C *(Bad)*

```
int processMessageFromSocket(int socket) {
  int success;
  char buffer[BUFFER_SIZE];
  char message[MESSAGE_SIZE];
  // get message from socket and store into buffer
  //Ignoring possibliity that buffer > BUFFER_SIZE
  if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
    // place contents of the buffer into message structure
    ExMessage *msg = recastBuffer(buffer);
    // copy message body into string for processing
    int index;
    for (index = 0; index < msg->msgLength; index++) {
      message[index] = msg->msgBody[index];
    }
    message[index] = '\0';
    // process message
    success = processMessage(message);
  }
  return success;
}
```

However, the message length variable from the structure is used as the condition for ending the for loop without validating that the message length variable accurately reflects the length of the message body (CWE-606). This can result in a buffer over-read (CWE-125) by reading from memory beyond the bounds of the buffer if the message length variable indicates a length that is longer than the size of a message body (CWE-130).

Observed Examples

Reference	Description
CVE-2014-0160	Chain: "Heartbleed" bug receives an inconsistent length parameter (CWE-130) enabling an out-of-bounds read (CWE-126), returning memory that could include private cryptographic keys and other sensitive data. *https://www.cve.org/CVERecord?id=CVE-2014-0160*
CVE-2009-2299	Web application firewall consumes excessive memory when an HTTP request contains a large Content-Length value but no POST data. *https://www.cve.org/CVERecord?id=CVE-2009-2299*
CVE-2001-0825	Buffer overflow in internal string handling routine allows remote attackers to execute arbitrary commands via a length argument of zero or less, which disables the length check. *https://www.cve.org/CVERecord?id=CVE-2001-0825*
CVE-2001-1186	Web server allows remote attackers to cause a denial of service via an HTTP request with a content-length value that is larger than the size of the request, which prevents server from timing out the connection. *https://www.cve.org/CVERecord?id=CVE-2001-1186*
CVE-2001-0191	Service does not properly check the specified length of a cookie, which allows remote attackers to execute arbitrary commands via a buffer overflow, or brute force authentication by using a short cookie length. *https://www.cve.org/CVERecord?id=CVE-2001-0191*
CVE-2003-0429	Traffic analyzer allows remote attackers to cause a denial of service and possibly execute arbitrary code via invalid IPv4 or IPv6 prefix lengths, possibly triggering a buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2003-0429*
CVE-2000-0655	Chat client allows remote attackers to cause a denial of service or execute arbitrary commands via a JPEG image containing a comment with an illegal field length of 1. *https://www.cve.org/CVERecord?id=CVE-2000-0655*
CVE-2004-0492	Server allows remote attackers to cause a denial of service and possibly execute arbitrary code via a negative Content-Length HTTP header field causing a heap-based buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2004-0492*
CVE-2004-0201	Help program allows remote attackers to execute arbitrary commands via a heap-based buffer overflow caused by a .CHM file with a large length field *https://www.cve.org/CVERecord?id=CVE-2004-0201*
CVE-2003-0825	Name services does not properly validate the length of certain packets, which allows attackers to cause a denial of service and possibly execute arbitrary code. Can overlap zero-length issues *https://www.cve.org/CVERecord?id=CVE-2003-0825*
CVE-2004-0095	Policy manager allows remote attackers to cause a denial of service (memory consumption and crash) and possibly execute arbitrary code via an HTTP POST request with an invalid Content-Length value. *https://www.cve.org/CVERecord?id=CVE-2004-0095*
CVE-2004-0826	Heap-based buffer overflow in library allows remote attackers to execute arbitrary code via a modified record length field in an SSLv2 client hello message.

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2004-0826
CVE-2004-0808	When domain logons are enabled, server allows remote attackers to cause a denial of service via a SAM_UAS_CHANGE request with a length value that is larger than the number of structures that are provided. *https://www.cve.org/CVERecord?id=CVE-2004-0808*
CVE-2002-1357	Multiple SSH2 servers and clients do not properly handle packets or data elements with incorrect length specifiers, which may allow remote attackers to cause a denial of service or possibly execute arbitrary code. *https://www.cve.org/CVERecord?id=CVE-2002-1357*
CVE-2004-0774	Server allows remote attackers to cause a denial of service (CPU and memory exhaustion) via a POST request with a Content-Length header set to -1. *https://www.cve.org/CVERecord?id=CVE-2004-0774*
CVE-2004-0989	Multiple buffer overflows in xml library that may allow remote attackers to execute arbitrary code via long URLs. *https://www.cve.org/CVERecord?id=CVE-2004-0989*
CVE-2004-0568	Application does not properly validate the length of a value that is saved in a session file, which allows remote attackers to execute arbitrary code via a malicious session file (.ht), web site, or Telnet URL contained in an e-mail message, triggering a buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2004-0568*
CVE-2003-0327	Server allows remote attackers to cause a denial of service via a remote password array with an invalid length, which triggers a heap-based buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2003-0327*
CVE-2003-0345	Product allows remote attackers to cause a denial of service and possibly execute arbitrary code via an SMB packet that specifies a smaller buffer length than is required. *https://www.cve.org/CVERecord?id=CVE-2003-0345*
CVE-2004-0430	Server allows remote attackers to execute arbitrary code via a LoginExt packet for a Cleartext Password User Authentication Method (UAM) request with a PathName argument that includes an AFPName type string that is longer than the associated length field. *https://www.cve.org/CVERecord?id=CVE-2004-0430*
CVE-2005-0064	PDF viewer allows remote attackers to execute arbitrary code via a PDF file with a large /Encrypt /Length keyLength value. *https://www.cve.org/CVERecord?id=CVE-2005-0064*
CVE-2004-0413	SVN client trusts the length field of SVN protocol URL strings, which allows remote attackers to cause a denial of service and possibly execute arbitrary code via an integer overflow that leads to a heap-based buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2004-0413*
CVE-2004-0940	Is effectively an accidental double increment of a counter that prevents a length check conditional from exiting a loop. *https://www.cve.org/CVERecord?id=CVE-2004-0940*
CVE-2002-1235	Length field of a request not verified. *https://www.cve.org/CVERecord?id=CVE-2002-1235*
CVE-2005-3184	Buffer overflow by modifying a length value. *https://www.cve.org/CVERecord?id=CVE-2005-3184*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

This probably overlaps other categories including zero-length issues.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Length Parameter Inconsistency
Software Fault Patterns	SFP24		Tainted Input to Command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
47	Buffer Overflow via Parameter Expansion

CWE-131: Incorrect Calculation of Buffer Size

Weakness ID : 131
Structure : Simple
Abstraction : Base

Description

The product does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499
ParentOf	V	467	Use of sizeof() on a Pointer Type	1110
CanPrecede	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
MemberOf	C	1218	Memory Buffer Errors	2479

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Availability Confidentiality	DoS: Crash, Exit, or Restart Execute Unauthorized Code or Commands Read Memory Modify Memory *If the incorrect calculation is used in the context of memory allocation, then the software may create a buffer that is smaller or larger than expected. If the allocated buffer is smaller than expected, this could lead to an out-of-bounds read or write (CWE-119), possibly causing a crash, allowing arbitrary code execution, or exposing sensitive data.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives. Automated static analysis generally does not account for environmental considerations when reporting potential errors in buffer calculations. This can make it difficult for users to determine which warnings should be investigated first. For example, an analysis tool might report buffer overflows that originate from command line arguments in a program that is not expected to run with setuid or other special privileges.

Effectiveness = High

Detection techniques for buffer-related errors are more mature than for most other weakness types.

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Effectiveness = Moderate

Without visibility into the code, black box methods may not be able to sufficiently distinguish this weakness from others, requiring follow-up manual methods to diagnose the underlying problem.

Manual Analysis

Manual analysis can be useful for finding this weakness, but it might not achieve desired code coverage within limited time constraints. This becomes difficult for weaknesses that must be considered for all inputs, since the attack surface can be too large.

Manual Analysis

This weakness can be detected using tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. Specifically, manual static analysis is useful for evaluating the correctness of allocation calculations. This can be useful for detecting overflow conditions (CWE-190) or similar weaknesses that might have serious security impacts on the program.

Effectiveness = High

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

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Bytecode Weakness Analysis - including disassembler + source code weakness analysis Binary Weakness Analysis - including disassembler + source code weakness analysis

Effectiveness = High

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source Manual Source Code Review (not inspections)

Effectiveness = SOAR Partial

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 Cost effective for partial coverage: Source Code Quality Analyzer

Effectiveness = High

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

When allocating a buffer for the purpose of transforming, converting, or encoding an input, allocate enough memory to handle the largest possible encoding. For example, in a routine that converts "&" characters to "&amp;" for HTML entity encoding, the output buffer needs to be at least 5 times as large as the input buffer.

Phase: Implementation

Understand the programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how the language handles numbers that are too large or too small for its underlying representation. [REF-7] Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation.

Phase: Implementation

Strategy = Input Validation

Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.

Phase: 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.

Phase: Implementation

When processing structured incoming data containing a size field followed by raw data, identify and resolve any inconsistencies between the size field and the actual size of the data (CWE-130).

Phase: Implementation

When allocating memory that uses sentinels to mark the end of a data structure - such as NUL bytes in strings - make sure you also include the sentinel in your calculation of the total amount of memory that must be allocated.

Phase: Implementation

Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.

Effectiveness = Moderate

This approach is still susceptible to calculation errors, including issues such as off-by-one errors (CWE-193) and incorrectly calculating buffer lengths (CWE-131). Additionally, this only addresses potential overflow issues. Resource consumption / exhaustion issues are still possible.

Phase: Implementation

Use sizeof() on the appropriate data type to avoid CWE-467.

Phase: Implementation

Use the appropriate type for the desired action. For example, in C/C++, only use unsigned types for values that could never be negative, such as height, width, or other numbers related to quantity. This will simplify validation and will reduce surprises related to unexpected casting.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Use libraries or frameworks that make it easier to handle numbers without unexpected consequences, or buffer allocation routines that automatically track buffer size. Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++). [REF-106]

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Use automatic buffer overflow detection mechanisms that are offered by certain compilers or compiler extensions. Examples include: the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND [REF-1334] discusses canary-based detection in detail.

Effectiveness = Defense in Depth

This is not necessarily a complete solution, since these mechanisms only detect certain types of overflows. In addition, the result is still a denial of service, since the typical response is to exit the application.

Phase: Operation

Phase: Build and Compilation

Strategy = Environment Hardening

Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335].

Effectiveness = Defense in Depth

These techniques do not provide a complete solution. For instance, exploits frequently use a bug that discloses memory addresses in order to maximize reliability of code execution [REF-1337]. It has also been shown that a side-channel attack can bypass ASLR [REF-1333]

Phase: Operation

Strategy = Environment Hardening

Use a CPU and operating system that offers Data Execution Protection (using hardware NX or XD bits) or the equivalent techniques that simulate this feature in software, such as PaX [REF-60] [REF-61]. These techniques ensure that any instruction executed is exclusively at a memory address that is part of the code segment. For more information on these techniques see D3-PSEP (Process Segment Execution Prevention) from D3FEND [REF-1336].

Effectiveness = Defense in Depth

This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Phase: Implementation

Strategy = Compilation or Build Hardening

Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.

Phase: Architecture and Design

Phase: Operation

Strategy = Environment Hardening

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.

Phase: Architecture and Design

Phase: Operation

Strategy = Sandbox or Jail

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

The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Demonstrative Examples

Example 1:

The following code allocates memory for a maximum number of widgets. It then gets a user-specified number of widgets, making sure that the user does not request too many. It then initializes the elements of the array using InitializeWidget(). Because the number of widgets can vary for each request, the code inserts a NULL pointer to signify the location of the last widget.

Example Language: C *(Bad)*

```c
int i;
unsigned int numWidgets;
Widget **WidgetList;
numWidgets = GetUntrustedSizeValue();
if ((numWidgets == 0) || (numWidgets > MAX_NUM_WIDGETS)) {
  ExitError("Incorrect number of widgets requested!");
}
WidgetList = (Widget **)malloc(numWidgets * sizeof(Widget *));
printf("WidgetList ptr=%p\n", WidgetList);
for(i=0; i<numWidgets; i++) {
  WidgetList[i] = InitializeWidget();
}
WidgetList[numWidgets] = NULL;
showWidgets(WidgetList);
```

However, this code contains an off-by-one calculation error (CWE-193). It allocates exactly enough space to contain the specified number of widgets, but it does not include the space for the NULL pointer. As a result, the allocated buffer is smaller than it is supposed to be (CWE-131). So if the user ever requests MAX_NUM_WIDGETS, there is an out-of-bounds write (CWE-787) when the NULL is assigned. Depending on the environment and compilation settings, this could cause memory corruption.

Example 2:

The following image processing code allocates a table for images.

Example Language: C *(Bad)*

```
img_t table_ptr; /*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

This code intends to allocate a table of size num_imgs, however as num_imgs grows large, the calculation determining the size of the list will eventually overflow (CWE-190). This will result in a very small list to be allocated instead. If the subsequent code operates on the list as if it were num_imgs long, it may result in many types of out-of-bounds problems (CWE-119).

Example 3:

This example applies an encoding procedure to an input string and stores it into a buffer.

Example Language: C *(Bad)*

```
char * copy_input(char *user_supplied_string){
  int i, dst_index;
  char *dst_buf = (char*)malloc(4*sizeof(char) * MAX_SIZE);
  if ( MAX_SIZE <= strlen(user_supplied_string) ){
    die("user string too long, die evil hacker!");
  }
  dst_index = 0;
  for ( i = 0; i < strlen(user_supplied_string); i++ ){
    if( '&' == user_supplied_string[i] ){
      dst_buf[dst_index++] = '&';
      dst_buf[dst_index++] = 'a';
      dst_buf[dst_index++] = 'm';
      dst_buf[dst_index++] = 'p';
      dst_buf[dst_index++] = ';';
    }
    else if ('<' == user_supplied_string[i] ){
      /* encode to &lt; */
    }
    else dst_buf[dst_index++] = user_supplied_string[i];
  }
  return dst_buf;
}
```

The programmer attempts to encode the ampersand character in the user-controlled string, however the length of the string is validated before the encoding procedure is applied. Furthermore, the programmer assumes encoding expansion will only expand a given character by a factor of 4, while the encoding of the ampersand expands by 5. As a result, when the encoding procedure expands the string it is possible to overflow the destination buffer if the attacker provides a string of many ampersands.

Example 4:

The following code is intended to read an incoming packet from a socket and extract one or more headers.

Example Language: C *(Bad)*

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;
sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;
if (numHeaders > 100) {
  ExitError("too many headers!");
}
```

```
headers = malloc(numHeaders * sizeof(PacketHeader);
ParsePacketHeaders(packet, headers);
```

The code performs a check to make sure that the packet does not contain too many headers. However, numHeaders is defined as a signed int, so it could be negative. If the incoming packet specifies a value such as -3, then the malloc calculation will generate a negative number (say, -300 if each header can be a maximum of 100 bytes). When this result is provided to malloc(), it is first converted to a size_t type. This conversion then produces a large value such as 4294966996, which may cause malloc() to fail or to allocate an extremely large amount of memory (CWE-195). With the appropriate negative numbers, an attacker could trick malloc() into using a very small positive number, which then allocates a buffer that is much smaller than expected, potentially leading to a buffer overflow.

Example 5:

The following code attempts to save three different identification numbers into an array. The array is allocated from memory using a call to malloc().

Example Language: C *(Bad)*

```
int *id_sequence;
/* Allocate space for an array of three ids. */
id_sequence = (int*) malloc(3);
if (id_sequence == NULL) exit(1);
/* Populate the id array. */
id_sequence[0] = 13579;
id_sequence[1] = 24680;
id_sequence[2] = 97531;
```

The problem with the code above is the value of the size parameter used during the malloc() call. It uses a value of '3' which by definition results in a buffer of three bytes to be created. However the intention was to create a buffer that holds three ints, and in C, each int requires 4 bytes worth of memory, so an array of 12 bytes is needed, 4 bytes for each int. Executing the above code could result in a buffer overflow as 12 bytes of data is being saved into 3 bytes worth of allocated space. The overflow would occur during the assignment of id_sequence[0] and would continue with the assignment of id_sequence[1] and id_sequence[2].

The malloc() call could have used '3*sizeof(int)' as the value for the size parameter in order to allocate the correct amount of space required to store the three ints.

Observed Examples

Reference	Description
CVE-2020-17087	Chain: integer truncation (CWE-197) causes small buffer allocation (CWE-131) leading to out-of-bounds write (CWE-787) in kernel pool, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2020-17087*
CVE-2004-1363	substitution overflow: buffer overflow using environment variables that are expanded after the length check is performed *https://www.cve.org/CVERecord?id=CVE-2004-1363*
CVE-2004-0747	substitution overflow: buffer overflow using expansion of environment variables *https://www.cve.org/CVERecord?id=CVE-2004-0747*
CVE-2005-2103	substitution overflow: buffer overflow using a large number of substitution strings *https://www.cve.org/CVERecord?id=CVE-2005-2103*
CVE-2005-3120	transformation overflow: product adds extra escape characters to incoming data, but does not account for them in the buffer length *https://www.cve.org/CVERecord?id=CVE-2005-3120*
CVE-2003-0899	transformation overflow: buffer overflow when expanding ">" to "&gt;", etc. *https://www.cve.org/CVERecord?id=CVE-2003-0899*

Reference	Description
CVE-2001-0334	expansion overflow: buffer overflow using wildcards *https://www.cve.org/CVERecord?id=CVE-2001-0334*
CVE-2001-0248	expansion overflow: long pathname + glob = overflow *https://www.cve.org/CVERecord?id=CVE-2001-0248*
CVE-2001-0249	expansion overflow: long pathname + glob = overflow *https://www.cve.org/CVERecord?id=CVE-2001-0249*
CVE-2002-0184	special characters in argument are not properly expanded *https://www.cve.org/CVERecord?id=CVE-2002-0184*
CVE-2004-0434	small length value leads to heap overflow *https://www.cve.org/CVERecord?id=CVE-2004-0434*
CVE-2002-1347	multiple variants *https://www.cve.org/CVERecord?id=CVE-2002-1347*
CVE-2005-0490	needs closer investigation, but probably expansion-based *https://www.cve.org/CVERecord?id=CVE-2005-0490*
CVE-2004-0940	needs closer investigation, but probably expansion-based *https://www.cve.org/CVERecord?id=CVE-2004-0940*
CVE-2008-0599	Chain: Language interpreter calculates wrong buffer size (CWE-131) by using "size = ptr ? X : Y" instead of "size = (ptr ? X : Y)" expression. *https://www.cve.org/CVERecord?id=CVE-2008-0599*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	802	2010 Top 25 - Risky Resource Management	800	2354
MemberOf	C	865	2011 Top 25 - Risky Resource Management	900	2371
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	974	SFP Secondary Cluster: Incorrect Buffer Length Computation	888	2406
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1162	SEI CERT C Coding Standard - Guidelines 08. Memory Management (MEM)	1154	2458
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Maintenance

This is a broad category. Some examples include: simple math errors, incorrectly updating parallel counters, not accounting for size differences when "transforming" one input to another format (e.g. URL canonicalization or other transformation that can generate a result that's larger than the original input, i.e. "expansion"). This level of detail is rarely available in public reports, so it is difficult to find good examples.

Maintenance

This weakness may be a composite or a chain. It also may contain layering or perspective differences. This issue may be associated with many different types of incorrect calculations (CWE-682), although the integer overflow (CWE-190) is probably the most prevalent. This can be primary to resource consumption problems (CWE-400), including uncontrolled memory allocation (CWE-789). However, its relationship with out-of-bounds buffer access (CWE-119) must also be considered.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Other length calculation error
CERT C Secure Coding	INT30-C	Imprecise	Ensure that unsigned integer operations do not wrap
CERT C Secure Coding	MEM35-C	CWE More Abstract	Allocate sufficient memory for an object

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
47	Buffer Overflow via Parameter Expansion
100	Overflow Buffers

References

[REF-106]David LeBlanc and Niels Dekker. "SafeInt". < http://safeint.codeplex.com/ >.

[REF-107]Jason Lam. "Top 25 Series - Rank 18 - Incorrect Calculation of Buffer Size". 2010 March 9. SANS Software Security Institute. < http://software-security.sans.org/blog/2010/03/19/top-25-series-rank-18-incorrect-calculation-of-buffer-size >.

[REF-58]Michael Howard. "Address Space Layout Randomization in Windows Vista". < https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista >.2023-04-07.

[REF-61]Microsoft. "Understanding DEP as a mitigation technology part 1". < https://msrc.microsoft.com/blog/2009/06/understanding-dep-as-a-mitigation-technology-part-1/ >.2023-04-07.

[REF-60]"PaX". < https://en.wikipedia.org/wiki/Executable_space_protection#PaX >.2023-04-07.

[REF-76]Sean Barnum and Michael Gegick. "Least Privilege". 2005 September 4. < https://web.archive.org/web/20211209014121/https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege >.2023-04-07.

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-64]Grant Murphy. "Position Independent Executables (PIE)". 2012 November 8. Red Hat. < https://www.redhat.com/en/blog/position-independent-executables-pie >.2023-04-07.

[REF-1332]John Richard Moser. "Prelink and address space randomization". 2006 July 5. < https://lwn.net/Articles/190139/ >.2023-04-26.

[REF-1333]Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". 2016. < http://www.cs.ucr.edu/~nael/pubs/micro16.pdf >.2023-04-26.

[REF-1334]D3FEND. "Stack Frame Canary Validation (D3-SFCV)". 2023. < https://d3fend.mitre.org/technique/d3f:StackFrameCanaryValidation/ >.2023-04-26.

[REF-1335]D3FEND. "Segment Address Offset Randomization (D3-SAOR)". 2023. < https://d3fend.mitre.org/technique/d3f:SegmentAddressOffsetRandomization/ >.2023-04-26.

[REF-1336]D3FEND. "Process Segment Execution Prevention (D3-PSEP)". 2023. < https://d3fend.mitre.org/technique/d3f:ProcessSegmentExecutionPrevention/ >.2023-04-26.

[REF-1337]Alexander Sotirov and Mark Dowd. "Bypassing Browser Memory Protections: Setting back browser security by 10 years". 2008. < https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf >.2023-04-26.

CWE-134: Use of Externally-Controlled Format String

Weakness ID : 134
Structure : Simple
Abstraction : Base

Description

The product uses a function that accepts a format string as an argument, but the format string originates from an external source.

Extended Description

When an attacker can modify an externally-controlled format string, this can lead to buffer overflows, denial of service, or data representation problems.

It should be noted that in some circumstances, such as internationalization, the set of format strings is externally controlled by design. If the source of these format strings is trusted (e.g. only contained in library files that are only modifiable by the system administrator), then the external control might not itself pose a vulnerability.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469
CanPrecede	B	123	Write-what-where Condition	323

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469

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

Nature	Type	ID	Name	Page
MemberOf	C	133	String Errors	2310

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Often)*

Language : C++ *(Prevalence = Often)*

Language : Perl *(Prevalence = Rarely)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Memory *Format string problems allow for information disclosure which can severely simplify exploitation of the program.*	
Integrity Confidentiality Availability	Modify Memory Execute Unauthorized Code or Commands *Format string problems can result in the execution of arbitrary code.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives.

Black Box

Since format strings often occur in rarely-occurring erroneous conditions (e.g. for error message logging), they can be difficult to detect using black box methods. It is highly likely that many latent issues exist in executables that do not have associated source code (or equivalent source.

Effectiveness = Limited

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Bytecode Weakness Analysis - including disassembler + source code weakness analysis Binary Weakness Analysis - including disassembler + source code weakness analysis Cost effective for partial coverage: Binary / Bytecode simple extractor - strings, ELF readers, etc.

Effectiveness = High

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies

Effectiveness = SOAR Partial

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Web Application Scanner Web Services Scanner Database Scanners

Effectiveness = SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Fuzz Tester Framework-based Fuzzer

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Manual Source Code Review (not inspections) Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source

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 Cost effective for partial coverage: Warning Flags

Effectiveness = High

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Requirements

Choose a language that is not subject to this flaw.

Phase: Implementation

Ensure that all format string functions are passed a static string which cannot be controlled by the user, and that the proper number of arguments are always sent to that function as well. If at all possible, use functions that do not support the %n operator in format strings. [REF-116] [REF-117]

Phase: Build and Compilation

Run compilers and linkers with high warning levels, since they may detect incorrect usage.

Demonstrative Examples

Example 1:

The following program prints a string provided as an argument.

Example Language: C *(Bad)*

```
#include <stdio.h>
void printWrapper(char *string) {
  printf(string);
}
int main(int argc, char **argv) {
  char buf[5012];
  memcpy(buf, argv[1], 5012);
  printWrapper(argv[1]);
  return (0);
}
```

The example is exploitable, because of the call to printf() in the printWrapper() function. Note: The stack buffer was added to make exploitation more simple.

Example 2:

The following code copies a command line argument into a buffer using snprintf().

Example Language: C *(Bad)*

```
int main(int argc, char **argv){
  char buf[128];
  ...
```

```
    snprintf(buf,128,argv[1]);
}
```

This code allows an attacker to view the contents of the stack and write to the stack using a command line argument containing a sequence of formatting directives. The attacker can read from the stack by providing more formatting directives, such as %x, than the function takes as arguments to be formatted. (In this example, the function takes no arguments to be formatted.) By using the %n formatting directive, the attacker can write to the stack, causing snprintf() to write the number of bytes output thus far to the specified argument (rather than reading a value from the argument, which is the intended behavior). A sophisticated version of this attack will use four staggered writes to completely control the value of a pointer on the stack.

Example 3:

Certain implementations make more advanced attacks even easier by providing format directives that control the location in memory to read from or write to. An example of these directives is shown in the following code, written for glibc:

Example Language: C *(Bad)*

```
printf("%d %d %1$d %1$d\n", 5, 9);
```

This code produces the following output: 5 9 5 5 It is also possible to use half-writes (%hn) to accurately control arbitrary DWORDS in memory, which greatly reduces the complexity needed to execute an attack that would otherwise require four staggered writes, such as the one mentioned in the first example.

Observed Examples

Reference	Description
CVE-2002-1825	format string in Perl program *https://www.cve.org/CVERecord?id=CVE-2002-1825*
CVE-2001-0717	format string in bad call to syslog function *https://www.cve.org/CVERecord?id=CVE-2001-0717*
CVE-2002-0573	format string in bad call to syslog function *https://www.cve.org/CVERecord?id=CVE-2002-0573*
CVE-2002-1788	format strings in NNTP server responses *https://www.cve.org/CVERecord?id=CVE-2002-1788*
CVE-2006-2480	Format string vulnerability exploited by triggering errors or warnings, as demonstrated via format string specifiers in a .bmp filename. *https://www.cve.org/CVERecord?id=CVE-2006-2480*
CVE-2007-2027	Chain: untrusted search path enabling resultant format string by loading malicious internationalization messages *https://www.cve.org/CVERecord?id=CVE-2007-2027*

Functional Areas

- Logging
- Error Handling
- String Processing

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	635	Weaknesses Originally Used by NVD from 2008 to 2016	635	2552
MemberOf	C	726	OWASP Top Ten 2004 Category A5 - Buffer Overflows	711	2336
MemberOf	C	743	CERT C Secure Coding Standard (2008) Chapter 10 - Input Output (FIO)	734	2347
MemberOf	C	808	2010 Top 25 - Weaknesses On the Cusp	800	2355
MemberOf	C	845	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 2 - Input Validation and Data Sanitization (IDS)	844	2362
MemberOf	C	865	2011 Top 25 - Risky Resource Management	900	2371
MemberOf	C	877	CERT C++ Secure Coding Section 09 - Input Output (FIO)	868	2377
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1131	CISQ Quality Measures (2016) - Security	1128	2442
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1163	SEI CERT C Coding Standard - Guidelines 09. Input Output (FIO)	1154	2459
MemberOf	C	1179	SEI CERT Perl Coding Standard - Guidelines 01. Input Validation and Data Sanitization (IDS)	1178	2465
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Applicable Platform

This weakness is possible in any programming language that support format strings.

Other

While Format String vulnerabilities typically fall under the Buffer Overflow category, technically they are not overflowed buffers. The Format String vulnerability is fairly new (circa 1999) and stems from the fact that there is no realistic way for a function that takes a variable number of arguments to determine just how many arguments were passed in. The most common functions that take a variable number of arguments, including C-runtime functions, are the printf() family of calls. The Format String problem appears in a number of ways. A *printf() call without a format specifier is dangerous and can be exploited. For example, printf(input); is exploitable, while printf(y, input); is not exploitable in that context. The result of the first call, used incorrectly, allows for an attacker to be able to peek at stack memory since the input string will be used as the format specifier. The attacker can stuff the input string with format specifiers and begin reading stack values, since the remaining parameters will be pulled from the stack. Worst case, this improper use may give away enough control to allow an arbitrary value (or values in the case of an exploit program) to be written into the memory of the running program. Frequently targeted entities are file names, process names, identifiers. Format string problems are a classic C/C++ issue that are now rare due to the ease of discovery. One main reason format string vulnerabilities can be exploited is due to the %n operator. The %n operator will write the number of characters, which have been printed by the format string therefore far, to the memory pointed to by its argument. Through skilled creation of a format string, a malicious user may use values on the stack to create a write-what-where condition. Once this is achieved, they can execute arbitrary code. Other operators can be used as well; for example, a %9999s operator could also trigger a buffer overflow, or when used in file-formatting functions like fprintf, it can generate a much larger output than intended.

Research Gap

Format string issues are under-studied for languages other than C. Memory or disk consumption, control flow or variable alteration, and data corruption may result from format string exploitation in applications written in other languages such as Perl, PHP, Python, etc.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Format string vulnerability
7 Pernicious Kingdoms			Format String
CLASP			Format string problem
CERT C Secure Coding	FIO30-C	Exact	Exclude user input from format strings
CERT C Secure Coding	FIO47-C	CWE More Specific	Use valid format strings
OWASP Top Ten 2004	A1	CWE More Specific	Unvalidated Input
WASC	6		Format String
The CERT Oracle Secure Coding Standard for Java (2011)	IDS06-J		Exclude user input from format strings
SEI CERT Perl Coding Standard	IDS30-PL	Exact	Exclude user input from format strings
Software Fault Patterns	SFP24		Tainted input to command
OMG ASCSM	ASCSM-CWE-134		

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
67	String Format Overflow in syslog()
135	Format String Injection

References

[REF-116]Steve Christey. "Format String Vulnerabilities in Perl Programs". < https://seclists.org/fulldisclosure/2005/Dec/91 >.2023-04-07.

[REF-117]Hal Burch and Robert C. Seacord. "Programming Language Format String Vulnerabilities". < https://drdobbs.com/security/programming-language-format-string-vulne/197002914 >.2023-04-07.

[REF-118]Tim Newsham. "Format String Attacks". 2000 September 9. Guardent. < http://www.thenewsh.com/~newsham/format-string-attacks.pdf >.

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

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

CWE-135: Incorrect Calculation of Multi-Byte String Length

Weakness ID : 135
Structure : Simple
Abstraction : Base

Description

The product does not correctly calculate the length of strings that can contain wide or multi-byte characters.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
MemberOf	C	133	String Errors	2310

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands *This weakness may lead to a buffer overflow. Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. This can often be used to subvert any other security service.*	
Availability Confidentiality	Read Memory DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) *Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.*	
Confidentiality	Read Memory *In the case of an out-of-bounds read, the attacker may have access to sensitive information. If the sensitive information contains system details, such as the current buffers position in memory, this knowledge can be used to craft further attacks, possibly with more severe consequences.*	

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

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Always verify the length of the string unit character.

Phase: Implementation

Strategy = Libraries or Frameworks

Use length computing functions (e.g. strlen, wcslen, etc.) appropriately with their equivalent type (e.g.: byte, wchar_t, etc.)

Demonstrative Examples

Example 1:

The following example would be exploitable if any of the commented incorrect malloc calls were used.

Example Language: C *(Bad)*

```
#include <stdio.h>
#include <strings.h>
#include <wchar.h>
int main() {
  wchar_t wideString[] = L"The spazzy orange tiger jumped " \
  "over the tawny jaguar.";
  wchar_t *newString;
  printf("Strlen() output: %d\nWcslen() output: %d\n",
  strlen(wideString), wcslen(wideString));
  /* Wrong because the number of chars in a string isn't related to its length in bytes //
  newString = (wchar_t *) malloc(strlen(wideString));
  */
  /* Wrong because wide characters aren't 1 byte long! //
  newString = (wchar_t *) malloc(wcslen(wideString));
  */
  /* Wrong because wcslen does not include the terminating null */
  newString = (wchar_t *) malloc(wcslen(wideString) * sizeof(wchar_t));
  /* correct! */
  newString = (wchar_t *) malloc((wcslen(wideString) + 1) * sizeof(wchar_t));
  /* ... */
}
```

The output from the printf() statement would be:

Example Language: *(Result)*

```
Strlen() output: 0
Wcslen() output: 53
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	741	CERT C Secure Coding Standard (2008) Chapter 8 - Characters and Strings (STR)	734	2344

Nature	Type	ID	Name	V	Page
MemberOf	C	857	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 14 - Input Output (FIO)	844	2368
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	974	SFP Secondary Cluster: Incorrect Buffer Length Computation	888	2406
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Improper string length checking
The CERT Oracle Secure Coding Standard for Java (2011)	FIO10-J		Ensure the array is filled when using read() to fill an array
Software Fault Patterns	SFP10		Incorrect Buffer Length Computation

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-138: Improper Neutralization of Special Elements

Weakness ID : 138
Structure : Simple
Abstraction : Class

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as control elements or syntactic markers when they are sent to a downstream component.

Extended Description

Most languages and protocols have their own special elements such as characters and reserved words. These special elements can carry control implications. If product does not prevent external control or influence over the inclusion of such special elements, the control flow of the program may be altered from what was intended. For example, both Unix and Windows interpret the symbol < ("less than") as meaning "read input from a file".

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	707	Improper Neutralization	1546
ParentOf	B	140	Improper Neutralization of Delimiters	376
ParentOf	V	147	Improper Neutralization of Input Terminators	389
ParentOf	V	148	Improper Neutralization of Input Leaders	391
ParentOf	V	149	Improper Neutralization of Quoting Syntax	392

Nature	Type	ID	Name	Page
ParentOf	V	150	Improper Neutralization of Escape, Meta, or Control Sequences	394
ParentOf	V	151	Improper Neutralization of Comment Delimiters	396
ParentOf	V	152	Improper Neutralization of Macro Symbols	398
ParentOf	V	153	Improper Neutralization of Substitution Characters	400
ParentOf	V	154	Improper Neutralization of Variable Name Delimiters	401
ParentOf	V	155	Improper Neutralization of Wildcards or Matching Symbols	403
ParentOf	V	156	Improper Neutralization of Whitespace	405
ParentOf	V	157	Failure to Sanitize Paired Delimiters	407
ParentOf	V	158	Improper Neutralization of Null Byte or NUL Character	409
ParentOf	C	159	Improper Handling of Invalid Use of Special Elements	411
ParentOf	V	160	Improper Neutralization of Leading Special Elements	413
ParentOf	V	162	Improper Neutralization of Trailing Special Elements	417
ParentOf	V	164	Improper Neutralization of Internal Special Elements	420
ParentOf	B	464	Addition of Data Structure Sentinel	1107
ParentOf	C	790	Improper Filtering of Special Elements	1678

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1019	Validate Inputs	2433

Weakness Ordinalities

Primary :

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability Other	Execute Unauthorized Code or Commands Alter Execution Logic DoS: Crash, Exit, or Restart	

Potential Mitigations

Phase: Implementation

Developers should anticipate that special elements (e.g. delimiters, symbols) will be injected into input vectors of their product. One defense is to create an allowlist (e.g. a regular expression) that defines valid input according to the requirements specifications. Strictly filter any input that does not match against the allowlist. Properly encode your output, and quote any elements that have special meaning to the component with which you are communicating.

Phase: 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.

Phase: Implementation

Use and specify an appropriate output encoding to ensure that the special elements are well-defined. A normal byte sequence in one encoding could be a special element in another.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Observed Examples

Reference	Description
CVE-2001-0677	Read arbitrary files from mail client by providing a special MIME header that is internally used to store pathnames for attachments. *https://www.cve.org/CVERecord?id=CVE-2001-0677*
CVE-2000-0703	Setuid program does not cleanse special escape sequence before sending data to a mail program, causing the mail program to process those sequences. *https://www.cve.org/CVERecord?id=CVE-2000-0703*
CVE-2003-0020	Multi-channel issue. Terminal escape sequences not filtered from log files. *https://www.cve.org/CVERecord?id=CVE-2003-0020*
CVE-2003-0083	Multi-channel issue. Terminal escape sequences not filtered from log files. *https://www.cve.org/CVERecord?id=CVE-2003-0083*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

This weakness can be related to interpretation conflicts or interaction errors in intermediaries (such as proxies or application firewalls) when the intermediary's model of an endpoint does not account for protocol-specific special elements.

Relationship

See this entry's children for different types of special elements that have been observed at one point or another. However, it can be difficult to find suitable CVE examples. In an attempt to be complete, CWE includes some types that do not have any associated observed example.

Research Gap

This weakness is probably under-studied for proprietary or custom formats. It is likely that these issues are fairly common in applications that use their own custom format for configuration files, logs, meta-data, messaging, etc. They would only be found by accident or with a focused effort based on an understanding of the format.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Special Elements (Characters or Reserved Words)
PLOVER			Custom Special Character Injection
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
15	Command Delimiters
34	HTTP Response Splitting
105	HTTP Request Splitting

CWE-140: Improper Neutralization of Delimiters

Weakness ID : 140
Structure : Simple
Abstraction : Base

Description

The product does not neutralize or incorrectly neutralizes delimiters.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373
ParentOf	V	141	Improper Neutralization of Parameter/Argument Delimiters	378
ParentOf	V	142	Improper Neutralization of Value Delimiters	380
ParentOf	V	143	Improper Neutralization of Record Delimiters	381
ParentOf	V	144	Improper Neutralization of Line Delimiters	383
ParentOf	V	145	Improper Neutralization of Section Delimiters	385
ParentOf	V	146	Improper Neutralization of Expression/Command Delimiters	387

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

Nature	Type	ID	Name	Page
MemberOf	C	137	Data Neutralization Issues	2311

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Developers should anticipate that delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2003-0307	Attacker inserts field separator into input to specify admin privileges. *https://www.cve.org/CVERecord?id=CVE-2003-0307*
CVE-2000-0293	Multiple internal space, insufficient quoting - program does not use proper delimiter between values. *https://www.cve.org/CVERecord?id=CVE-2000-0293*
CVE-2001-0527	Attacker inserts carriage returns and "|" field separator characters to add new user/privileges. *https://www.cve.org/CVERecord?id=CVE-2001-0527*

Reference	Description
CVE-2002-0267	Linebreak in field of PHP script allows admin privileges when written to data file. *https://www.cve.org/CVERecord?id=CVE-2002-0267*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Delimiter Problems
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
15	Command Delimiters

CWE-141: Improper Neutralization of Parameter/Argument Delimiters

Weakness ID : 141
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as parameter or argument delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	140	Improper Neutralization of Delimiters	376

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that parameter/argument delimiters will be injected/removed/ manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2003-0307	Attacker inserts field separator into input to specify admin privileges. *https://www.cve.org/CVERecord?id=CVE-2003-0307*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Parameter Delimiter
Software Fault Patterns	SFP24		Tainted input to command

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-142: Improper Neutralization of Value Delimiters

Weakness ID : 142
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as value delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	140	Improper Neutralization of Delimiters	376

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that value delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2000-0293	Multiple internal space, insufficient quoting - program does not use proper delimiter between values. *https://www.cve.org/CVERecord?id=CVE-2000-0293*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Value Delimiter
Software Fault Patterns	SFP24		Tainted input to command

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-143: Improper Neutralization of Record Delimiters

Weakness ID : 143
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as record delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	140	Improper Neutralization of Delimiters	376

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that record delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2004-1982	Carriage returns in subject field allow adding new records to data file. *https://www.cve.org/CVERecord?id=CVE-2004-1982*
CVE-2001-0527	Attacker inserts carriage returns and "|" field separator characters to add new user/privileges. *https://www.cve.org/CVERecord?id=CVE-2001-0527*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Record Delimiter
Software Fault Patterns	SFP24		Tainted input to command

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-144: Improper Neutralization of Line Delimiters

Weakness ID : 144
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as line delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	140	Improper Neutralization of Delimiters	376

Nature	Type	ID	Name	Page
CanAlsoBe	B	93	Improper Neutralization of CRLF Sequences ('CRLF Injection')	217

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that line delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0267	Linebreak in field of PHP script allows admin privileges when written to data file. *https://www.cve.org/CVERecord?id=CVE-2002-0267*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	845	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 2 - Input Validation and Data Sanitization (IDS)	844	2362
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

Depending on the language and syntax being used, this could be the same as the record delimiter (CWE-143).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Line Delimiter
The CERT Oracle Secure Coding Standard for Java (2011)	IDS03-J		Do not log unsanitized user input
Software Fault Patterns	SFP24		Tainted input to command

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-145: Improper Neutralization of Section Delimiters

Weakness ID : 145
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as section delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

One example of a section delimiter is the boundary string in a multipart MIME message. In many cases, doubled line delimiters can serve as a section delimiter.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	140	Improper Neutralization of Delimiters	376
CanAlsoBe	B	93	Improper Neutralization of CRLF Sequences ('CRLF Injection')	217

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that section delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

Depending on the language and syntax being used, this could be the same as the record delimiter (CWE-143).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Section Delimiter
Software Fault Patterns	SFP24		Tainted input to command

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-146: Improper Neutralization of Expression/Command Delimiters

Weakness ID : 146
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as expression or command delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	140	Improper Neutralization of Delimiters	376

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability Other	Execute Unauthorized Code or Commands Alter Execution Logic	

Potential Mitigations

Developers should anticipate that inter-expression and inter-command delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

A shell metacharacter (covered in CWE-150) is one example of a potential delimiter that may need to be neutralized.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Delimiter between Expressions or Commands

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
6	Argument Injection
15	Command Delimiters

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-147: Improper Neutralization of Input Terminators

Weakness ID : 147
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as input terminators when they are sent to a downstream component.

Extended Description

For example, a "." in SMTP signifies the end of mail message data, whereas a null character can be used for the end of a string.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373
ParentOf	V	626	Null Byte Interaction Error (Poison Null Byte)	1394

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that terminators will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2000-0319	MFV. mail server does not properly identify terminator string to signify end of message, causing corruption, possibly in conjunction with off-by-one error. *https://www.cve.org/CVERecord?id=CVE-2000-0319*
CVE-2000-0320	MFV. mail server does not properly identify terminator string to signify end of message, causing corruption, possibly in conjunction with off-by-one error. *https://www.cve.org/CVERecord?id=CVE-2000-0320*
CVE-2001-0996	Mail server does not quote end-of-input terminator if it appears in the middle of a message. *https://www.cve.org/CVERecord?id=CVE-2001-0996*
CVE-2002-0001	Improperly terminated comment or phrase allows commands. *https://www.cve.org/CVERecord?id=CVE-2002-0001*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Input Terminator
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
460	HTTP Parameter Pollution (HPP)

CWE-148: Improper Neutralization of Input Leaders

Weakness ID : 148
Structure : Simple
Abstraction : Variant

Description

The product does not properly handle when a leading character or sequence ("leader") is missing or malformed, or if multiple leaders are used when only one should be allowed.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	138	Improper Neutralization of Special Elements	373

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that leading characters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Input Leader
Software Fault Patterns	SFP24		Tainted input to command

CWE-149: Improper Neutralization of Quoting Syntax

Weakness ID : 149
Structure : Simple
Abstraction : Variant

Description

Quotes injected into a product can be used to compromise a system. As data are parsed, an injected/absent/duplicate/malformed use of quotes may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that quotes will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2004-0956	Database allows remote attackers to cause a denial of service (application crash) via a MATCH AGAINST query with an opening double quote but no closing double quote. *https://www.cve.org/CVERecord?id=CVE-2004-0956*
CVE-2003-1016	MIE. MFV too? bypass AV/security with fields that should not be quoted, duplicate quotes, missing leading/trailing quotes. *https://www.cve.org/CVERecord?id=CVE-2003-1016*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Quoting Element

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
468	Generic Cross-Browser Cross-Domain Theft

CWE-150: Improper Neutralization of Escape, Meta, or Control Sequences

Weakness ID : 150
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as escape, meta, or control character sequences when they are sent to a downstream component.

Extended Description

As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	G	138	Improper Neutralization of Special Elements	373

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1019	Validate Inputs	2433

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that escape, meta and control characters/sequences will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0542	The mail program processes special "~" escape sequence even when not in interactive mode. *https://www.cve.org/CVERecord?id=CVE-2002-0542*
CVE-2000-0703	Setuid program does not filter escape sequences before calling mail program. *https://www.cve.org/CVERecord?id=CVE-2000-0703*
CVE-2002-0986	Mail function does not filter control characters from arguments, allowing mail message content to be modified. *https://www.cve.org/CVERecord?id=CVE-2002-0986*
CVE-2003-0020	Multi-channel issue. Terminal escape sequences not filtered from log files. *https://www.cve.org/CVERecord?id=CVE-2003-0020*
CVE-2003-0083	Multi-channel issue. Terminal escape sequences not filtered from log files. *https://www.cve.org/CVERecord?id=CVE-2003-0083*
CVE-2003-0021	Terminal escape sequences not filtered by terminals when displaying files. *https://www.cve.org/CVERecord?id=CVE-2003-0021*
CVE-2003-0022	Terminal escape sequences not filtered by terminals when displaying files. *https://www.cve.org/CVERecord?id=CVE-2003-0022*
CVE-2003-0023	Terminal escape sequences not filtered by terminals when displaying files. *https://www.cve.org/CVERecord?id=CVE-2003-0023*
CVE-2003-0063	Terminal escape sequences not filtered by terminals when displaying files. *https://www.cve.org/CVERecord?id=CVE-2003-0063*
CVE-2000-0476	Terminal escape sequences not filtered by terminals when displaying files. *https://www.cve.org/CVERecord?id=CVE-2000-0476*
CVE-2001-1556	MFV. (multi-channel). Injection of control characters into log files that allow information hiding when using raw Unix programs to read the files. *https://www.cve.org/CVERecord?id=CVE-2001-1556*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	845	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 2 - Input Validation and Data Sanitization (IDS)	844	2362
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Escape, Meta, or Control Character / Sequence
The CERT Oracle Secure Coding Standard for Java (2011)	IDS03-J		Do not log unsanitized user input
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
41	Using Meta-characters in E-mail Headers to Inject Malicious Payloads
81	Web Server Logs Tampering
93	Log Injection-Tampering-Forging
134	Email Injection

CWE-151: Improper Neutralization of Comment Delimiters

Weakness ID : 151
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as comment delimiters when they are sent to a downstream component.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that comments will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0001	Mail client command execution due to improperly terminated comment in address list. *https://www.cve.org/CVERecord?id=CVE-2002-0001*
CVE-2004-0162	MIE. RFC822 comment fields may be processed as other fields by clients. *https://www.cve.org/CVERecord?id=CVE-2004-0162*
CVE-2004-1686	Well-placed comment bypasses security warning. *https://www.cve.org/CVERecord?id=CVE-2004-1686*
CVE-2005-1909	Information hiding using a manipulation involving injection of comment code into product. Note: these vulnerabilities are likely vulnerable to more general

Reference	Description
	XSS problems, although a regexp might allow ">!--" while denying most other tags. *https://www.cve.org/CVERecord?id=CVE-2005-1909*
CVE-2005-1969	Information hiding using a manipulation involving injection of comment code into product. Note: these vulnerabilities are likely vulnerable to more general XSS problems, although a regexp might allow "<!--" while denying most other tags. *https://www.cve.org/CVERecord?id=CVE-2005-1969*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Comment Element
Software Fault Patterns	SFP24		Tainted input to command

CWE-152: Improper Neutralization of Macro Symbols

Weakness ID : 152
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as macro symbols when they are sent to a downstream component.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Developers should anticipate that macro symbols will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component.

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0770	Server trusts client to expand macros, allows macro characters to be expanded to trigger resultant information exposure. *https://www.cve.org/CVERecord?id=CVE-2002-0770*
CVE-2008-2018	Attacker can obtain sensitive information from a database by using a comment containing a macro, which inserts the data during expansion. *https://www.cve.org/CVERecord?id=CVE-2008-2018*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Research Gap

Under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Macro Symbol
Software Fault Patterns	SFP24		Tainted input to command

CWE-153: Improper Neutralization of Substitution Characters

Weakness ID : 153
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as substitution characters when they are sent to a downstream component.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that substitution characters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0770	Server trusts client to expand macros, allows macro characters to be expanded to trigger resultant information exposure. *https://www.cve.org/CVERecord?id=CVE-2002-0770*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Research Gap

Under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Substitution Character
Software Fault Patterns	SFP24		Tainted input to command

CWE-154: Improper Neutralization of Variable Name Delimiters

Weakness ID : 154
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as variable name delimiters when they are sent to a downstream component.

Extended Description

As data is parsed, an injected delimiter may cause the process to take unexpected actions that result in an attack. Example: "$" for an environment variable.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that variable name delimiters will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2005-0129	"%" variable is expanded by wildcard function into disallowed commands. *https://www.cve.org/CVERecord?id=CVE-2005-0129*
CVE-2002-0770	Server trusts client to expand macros, allows macro characters to be expanded to trigger resultant information exposure. *https://www.cve.org/CVERecord?id=CVE-2002-0770*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Research Gap

Under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Variable Name Delimiter
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
15	Command Delimiters

CWE-155: Improper Neutralization of Wildcards or Matching Symbols

Weakness ID : 155
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as wildcards or matching symbols when they are sent to a downstream component.

Extended Description

As data is parsed, an injected element may cause the process to take unexpected actions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	138	Improper Neutralization of Special Elements	373
ParentOf	Ⓥ	56	Path Equivalence: 'filedir*' (Wildcard)	107

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that wildcard or matching elements will be injected/removed/ manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0433	Bypass file restrictions using wildcard character. *https://www.cve.org/CVERecord?id=CVE-2002-0433*
CVE-2002-1010	Bypass file restrictions using wildcard character. *https://www.cve.org/CVERecord?id=CVE-2002-1010*
CVE-2001-0334	Wildcards generate long string on expansion. *https://www.cve.org/CVERecord?id=CVE-2001-0334*
CVE-2004-1962	SQL injection involving "/**/" sequences. *https://www.cve.org/CVERecord?id=CVE-2004-1962*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Research Gap

Under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Wildcard or Matching Element
Software Fault Patterns	SFP24		Tainted input to command

CWE-156: Improper Neutralization of Whitespace

Weakness ID : 156
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could be interpreted as whitespace when they are sent to a downstream component.

Extended Description

This can include space, tab, etc.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Alternate Terms

White space :

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that whitespace will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-0637	MIE. virus protection bypass with RFC violations involving extra whitespace, or missing whitespace. *https://www.cve.org/CVERecord?id=CVE-2002-0637*
CVE-2004-0942	CPU consumption with MIME headers containing lines with many space characters, probably due to algorithmic complexity (RESOURCE.AMP.ALG). *https://www.cve.org/CVERecord?id=CVE-2004-0942*
CVE-2003-1015	MIE. whitespace interpreted differently by mail clients. *https://www.cve.org/CVERecord?id=CVE-2003-1015*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

Can overlap other separator characters or delimiters.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER	SPEC.WHITESPACE		Whitespace
Software Fault Patterns	SFP24		Tainted input to command

CWE-157: Failure to Sanitize Paired Delimiters

Weakness ID : 157
Structure : Simple
Abstraction : Variant

Description

The product does not properly handle the characters that are used to mark the beginning and ending of a group of entities, such as parentheses, brackets, and braces.

Extended Description

Paired delimiters might include:

- < and > angle brackets
- (and) parentheses
- { and } braces
- [and] square brackets
- " " double quotes
- ' ' single quotes

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Scope	Impact	Likelihood

Potential Mitigations

Developers should anticipate that grouping elements will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2004-0956	Crash via missing paired delimiter (open double-quote but no closing double-quote). *https://www.cve.org/CVERecord?id=CVE-2004-0956*
CVE-2000-1165	Crash via message without closing ">". *https://www.cve.org/CVERecord?id=CVE-2000-1165*
CVE-2005-2933	Buffer overflow via mailbox name with an opening double quote but missing a closing double quote, causing a larger copy than expected. *https://www.cve.org/CVERecord?id=CVE-2005-2933*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Research Gap

Under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Grouping Element / Paired Delimiter
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
15	Command Delimiters

CWE-158: Improper Neutralization of Null Byte or NUL Character

Weakness ID : 158
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes NUL characters or null bytes when they are sent to a downstream component.

Extended Description

As data is parsed, an injected NUL character or null byte may cause the product to believe the input is terminated earlier than it actually is, or otherwise cause the input to be misinterpreted. This could then be used to inject potentially dangerous input that occurs after the null byte or otherwise bypass validation routines and other protection mechanisms.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	138	Improper Neutralization of Special Elements	373

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that null characters or null bytes will be injected/removed/ manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: 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.

Observed Examples

Reference	Description
CVE-2008-1284	NUL byte in theme name causes directory traversal impact to be worse *https://www.cve.org/CVERecord?id=CVE-2008-1284*
CVE-2005-2008	Source code disclosure using trailing null. *https://www.cve.org/CVERecord?id=CVE-2005-2008*
CVE-2005-3293	Source code disclosure using trailing null. *https://www.cve.org/CVERecord?id=CVE-2005-3293*
CVE-2005-2061	Trailing null allows file include. *https://www.cve.org/CVERecord?id=CVE-2005-2061*
CVE-2002-1774	Null character in MIME header allows detection bypass. *https://www.cve.org/CVERecord?id=CVE-2002-1774*
CVE-2000-0149	Web server allows remote attackers to view the source code for CGI programs via a null character (%00) at the end of a URL. *https://www.cve.org/CVERecord?id=CVE-2000-0149*
CVE-2000-0671	Web server earlier allows allows remote attackers to bypass access restrictions, list directory contents, and read source code by inserting a null character (%00) in the URL. *https://www.cve.org/CVERecord?id=CVE-2000-0671*
CVE-2001-0738	Logging system allows an attacker to cause a denial of service (hang) by causing null bytes to be placed in log messages. *https://www.cve.org/CVERecord?id=CVE-2001-0738*
CVE-2001-1140	Web server allows source code for executable programs to be read via a null character (%00) at the end of a request. *https://www.cve.org/CVERecord?id=CVE-2001-1140*
CVE-2002-1031	Protection mechanism for limiting file access can be bypassed using a null character (%00) at the end of the directory name. *https://www.cve.org/CVERecord?id=CVE-2002-1031*
CVE-2002-1025	Application server allows remote attackers to read JSP source code via an encoded null byte in an HTTP GET request, which causes the server to send the .JSP file unparsed. *https://www.cve.org/CVERecord?id=CVE-2002-1025*

Reference	Description
CVE-2003-0768	XSS protection mechanism only checks for sequences with an alphabetical character following a (<), so a non-alphabetical or null character (%00) following a < may be processed. *https://www.cve.org/CVERecord?id=CVE-2003-0768*
CVE-2004-0189	Decoding function in proxy allows regular expression bypass in ACLs via URLs with null characters. *https://www.cve.org/CVERecord?id=CVE-2004-0189*
CVE-2005-3153	Null byte bypasses PHP regexp check (interaction error). *https://www.cve.org/CVERecord?id=CVE-2005-3153*
CVE-2005-4155	Null byte bypasses PHP regexp check (interaction error). *https://www.cve.org/CVERecord?id=CVE-2005-4155*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

This can be a factor in multiple interpretation errors, other interaction errors, filename equivalence, etc.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Null Character / Null Byte
WASC	28		Null Byte Injection
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
52	Embedding NULL Bytes
53	Postfix, Null Terminate, and Backslash

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-159: Improper Handling of Invalid Use of Special Elements

Weakness ID : 159
Structure : Simple
Abstraction : Class

Description

The product does not properly filter, remove, quote, or otherwise manage the invalid use of special elements in user-controlled input, which could cause adverse effect on its behavior and integrity.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373
ParentOf	B	166	Improper Handling of Missing Special Element	423
ParentOf	B	167	Improper Handling of Additional Special Element	425
ParentOf	B	168	Improper Handling of Inconsistent Special Elements	426

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that special elements will be injected/removed/manipulated in the input vectors of their software system. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-1362	Crash via message type without separator character *https://www.cve.org/CVERecord?id=CVE-2002-1362*
CVE-2000-0116	Extra "<" in front of SCRIPT tag bypasses XSS prevention. *https://www.cve.org/CVERecord?id=CVE-2000-0116*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Maintenance

The list of children for this entry is far from complete. However, the types of special elements might be too precise for use within CWE.

Terminology

Precise terminology for the underlying weaknesses does not exist. Therefore, these weaknesses use the terminology associated with the manipulation.

Research Gap

Customized languages and grammars, even those that are specific to a particular product, are potential sources of weaknesses that are related to special elements. However, most researchers concentrate on the most commonly used representations for data transmission, such as HTML and SQL. Any representation that is commonly used is likely to be a rich source of weaknesses; researchers are encouraged to investigate previously unexplored representations.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Common Special Element Manipulations
Software Fault Patterns	SFP24		Tainted input to command

CWE-160: Improper Neutralization of Leading Special Elements

Weakness ID : 160
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes leading special elements that could be interpreted in unexpected ways when they are sent to a downstream component.

Extended Description

As data is parsed, improperly handled leading special elements may cause the process to take unexpected actions that result in an attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373
ParentOf	V	37	Path Traversal: '/absolute/pathname/here'	79
ParentOf	V	161	Improper Neutralization of Multiple Leading Special Elements	415

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that leading special elements will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-1345	Multiple FTP clients write arbitrary files via absolute paths in server responses *https://www.cve.org/CVERecord?id=CVE-2002-1345*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Leading Special Element
Software Fault Patterns	SFP24		Tainted input to command

CWE-161: Improper Neutralization of Multiple Leading Special Elements

Weakness ID : 161
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes multiple leading special elements that could be interpreted in unexpected ways when they are sent to a downstream component.

Extended Description

As data is parsed, improperly handled multiple leading special elements may cause the process to take unexpected actions that result in an attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	V	160	Improper Neutralization of Leading Special Elements	413
ParentOf	V	50	Path Equivalence: '//multiple/leading/slash'	100

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that multiple leading special elements will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-1238	Server allows remote attackers to bypass access restrictions for files via an HTTP request with a sequence of multiple / (slash) characters such as http://www.example.com///file/. *https://www.cve.org/CVERecord?id=CVE-2002-1238*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Multiple Leading Special Elements
Software Fault Patterns	SFP24		Tainted input to command

CWE-162: Improper Neutralization of Trailing Special Elements

Weakness ID : 162
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes trailing special elements that could be interpreted in unexpected ways when they are sent to a downstream component.

Extended Description

As data is parsed, improperly handled trailing special elements may cause the process to take unexpected actions that result in an attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	138	Improper Neutralization of Special Elements	373
ParentOf	Ⓥ	42	Path Equivalence: 'filename.' (Trailing Dot)	92
ParentOf	Ⓥ	46	Path Equivalence: 'filename ' (Trailing Space)	96
ParentOf	Ⓥ	49	Path Equivalence: 'filename/' (Trailing Slash)	99
ParentOf	Ⓥ	54	Path Equivalence: 'filedir\' (Trailing Backslash)	105
ParentOf	Ⓥ	163	Improper Neutralization of Multiple Trailing Special Elements	418

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that trailing special elements will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2004-0847	web framework for .NET allows remote attackers to bypass authentication for .aspx files in restricted directories via a request containing a (1) "\" (backslash) or (2) "%5C" (encoded backslash). *https://www.cve.org/CVERecord?id=CVE-2004-0847*
CVE-2002-1451	Trailing space ("+" in query string) leads to source code disclosure. *https://www.cve.org/CVERecord?id=CVE-2002-1451*
CVE-2001-0446	Application server allows remote attackers to read source code for .jsp files by appending a / to the requested URL. *https://www.cve.org/CVERecord?id=CVE-2001-0446*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Trailing Special Element
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
635	Alternative Execution Due to Deceptive Filenames

CWE-163: Improper Neutralization of Multiple Trailing Special Elements

Weakness ID : 163
Structure : Simple

Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes multiple trailing special elements that could be interpreted in unexpected ways when they are sent to a downstream component.

Extended Description

As data is parsed, improperly handled multiple trailing special elements may cause the process to take unexpected actions that result in an attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	V	162	Improper Neutralization of Trailing Special Elements	417
ParentOf	V	43	Path Equivalence: 'filename....' (Multiple Trailing Dot)	93
ParentOf	V	52	Path Equivalence: '/multiple/trailing/slash//'	103

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that multiple trailing special elements will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-1078	Directory listings in web server using multiple trailing slash *https://www.cve.org/CVERecord?id=CVE-2002-1078*
CVE-2004-0281	Multiple trailing dot allows directory listing *https://www.cve.org/CVERecord?id=CVE-2004-0281*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Multiple Trailing Special Elements
Software Fault Patterns	SFP24		Tainted input to command

CWE-164: Improper Neutralization of Internal Special Elements

Weakness ID : 164
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes internal special elements that could be interpreted in unexpected ways when they are sent to a downstream component.

Extended Description

As data is parsed, improperly handled internal special elements may cause the process to take unexpected actions that result in an attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	138	Improper Neutralization of Special Elements	373
ParentOf	V	165	Improper Neutralization of Multiple Internal Special Elements	422

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that internal special elements will be injected/removed/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413

Nature	Type	ID	Name	V	Page
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Internal Special Element
Software Fault Patterns	SFP24		Tainted input to command

CWE-165: Improper Neutralization of Multiple Internal Special Elements

Weakness ID : 165
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes multiple internal special elements that could be interpreted in unexpected ways when they are sent to a downstream component.

Extended Description

As data is parsed, improperly handled multiple internal special elements may cause the process to take unexpected actions that result in an attack.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	V	164	Improper Neutralization of Internal Special Elements	420
ParentOf	V	45	Path Equivalence: 'file...name' (Multiple Internal Dot)	95
ParentOf	V	53	Path Equivalence: '\multiple\\internal\backslash'	104

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that multiple internal special elements will be injected/removed/ manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Multiple Internal Special Element
Software Fault Patterns	SFP24		Tainted input to command

CWE-166: Improper Handling of Missing Special Element

Weakness ID : 166
Structure : Simple
Abstraction : Base

Description

The product receives input from an upstream component, but it does not handle or incorrectly handles when an expected special element is missing.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	228	Improper Handling of Syntactically Invalid Structure	568
ChildOf	●	159	Improper Handling of Invalid Use of Special Elements	411

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

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart	

Potential Mitigations

Developers should anticipate that special elements will be removed in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: 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.

Observed Examples

Reference	Description
CVE-2002-1362	Crash via message type without separator character *https://www.cve.org/CVERecord?id=CVE-2002-1362*
CVE-2002-0729	Missing special character (separator) causes crash *https://www.cve.org/CVERecord?id=CVE-2002-0729*
CVE-2002-1532	HTTP GET without \r\n\r\n CRLF sequences causes product to wait indefinitely and prevents other users from accessing it *https://www.cve.org/CVERecord?id=CVE-2002-1532*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Missing Special Element

CWE-167: Improper Handling of Additional Special Element

Weakness ID : 167
Structure : Simple
Abstraction : Base

Description

The product receives input from an upstream component, but it does not handle or incorrectly handles when an additional unexpected special element is provided.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	228	Improper Handling of Syntactically Invalid Structure	568
ChildOf	C	159	Improper Handling of Invalid Use of Special Elements	411

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

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Developers should anticipate that extra special elements will be injected in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2000-0116	Extra "<" in front of SCRIPT tag bypasses XSS prevention. *https://www.cve.org/CVERecord?id=CVE-2000-0116*
CVE-2001-1157	Extra "<" in front of SCRIPT tag. *https://www.cve.org/CVERecord?id=CVE-2001-1157*
CVE-2002-2086	"<script" - probably a cleansing error *https://www.cve.org/CVERecord?id=CVE-2002-2086*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Extra Special Element

CWE-168: Improper Handling of Inconsistent Special Elements

Weakness ID : 168

Structure : Simple
Abstraction : Base

Description

The product does not properly handle input in which an inconsistency exists between two or more special characters or reserved words.

Extended Description

An example of this problem would be if paired characters appear in the wrong order, or if the special characters are not properly nested.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	🟢	228	Improper Handling of Syntactically Invalid Structure	568
ChildOf	🟢	159	Improper Handling of Invalid Use of Special Elements	411

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

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Availability Access Control Non-Repudiation	DoS: Crash, Exit, or Restart Bypass Protection Mechanism Hide Activities	

Potential Mitigations

Developers should anticipate that inconsistent special elements will be injected/manipulated in the input vectors of their product. Use an appropriate combination of denylists and allowlists to ensure only valid, expected and appropriate input is processed by the system.

Phase: 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.

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Inconsistent Special Elements

CWE-170: Improper Null Termination

Weakness ID : 170
Structure : Simple
Abstraction : Base

Description

The product does not terminate or incorrectly terminates a string or array with a null character or equivalent terminator.

Extended Description

Null termination errors frequently occur in two different ways. An off-by-one error could cause a null to be written out of bounds, leading to an overflow. Or, a program could use a strncpy() function call incorrectly, which prevents a null terminator from being added at all. Other scenarios are possible.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	707	Improper Neutralization	1546
PeerOf	B	463	Deletion of Data Structure Sentinel	1105
PeerOf	B	464	Addition of Data Structure Sentinel	1107
CanAlsoBe	V	147	Improper Neutralization of Input Terminators	389
CanFollow	B	193	Off-by-one Error	486
CanFollow	P	682	Incorrect Calculation	1499
CanPrecede	B	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
CanPrecede	V	126	Buffer Over-read	334

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

Nature	Type	ID	Name	Page
MemberOf	C	137	Data Neutralization Issues	2311

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	●	20	Improper Input Validation	20

Weakness Ordinalities

Resultant :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Read Memory Execute Unauthorized Code or Commands *The case of an omitted null character is the most dangerous of the possible issues. This will almost certainly result in information disclosure, and possibly a buffer overflow condition, which may be exploited to execute arbitrary code.*	
Confidentiality Integrity Availability	DoS: Crash, Exit, or Restart Read Memory DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) *If a null character is omitted from a string, then most string-copying functions will read data until they locate a null character, even outside of the intended boundaries of the string. This could: cause a crash due to a segmentation fault cause sensitive adjacent memory to be copied and sent to an outsider trigger a buffer overflow when the copy is being written to a fixed-size buffer.*	
Integrity Availability	Modify Memory DoS: Crash, Exit, or Restart *Misplaced null characters may result in any number of security problems. The biggest issue is a subset of buffer overflow, and write-what-where conditions, where data corruption occurs from the writing of a null character over valid data, or even instructions. A randomly placed null character may put the system into an undefined state, and therefore make it prone to crashing. A misplaced null character may corrupt other data in memory.*	
Integrity Confidentiality Availability Access Control Other	Alter Execution Logic Execute Unauthorized Code or Commands *Should the null character corrupt the process flow, or affect a flag controlling access, it may lead to logical errors which allow for the execution of arbitrary code.*	

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

Potential Mitigations

Phase: Requirements

Use a language that is not susceptible to these issues. However, be careful of null byte interaction errors (CWE-626) with lower-level constructs that may be written in a language that is susceptible.

Phase: Implementation

Ensure that all string functions used are understood fully as to how they append null characters. Also, be wary of off-by-one errors when appending nulls to the end of strings.

Phase: Implementation

If performance constraints permit, special code can be added that validates null-termination of string buffers, this is a rather naive and error-prone solution.

Phase: Implementation

Switch to bounded string manipulation functions. Inspect buffer lengths involved in the buffer overrun trace reported with the defect.

Phase: Implementation

Add code that fills buffers with nulls (however, the length of buffers still needs to be inspected, to ensure that the non null-terminated string is not written at the physical end of the buffer).

Demonstrative Examples

Example 1:

The following code reads from cfgfile and copies the input into inputbuf using strcpy(). The code mistakenly assumes that inputbuf will always contain a NULL terminator.

Example Language: C *(Bad)*

```c
#define MAXLEN 1024
...
char *pathbuf[MAXLEN];
...
read(cfgfile,inputbuf,MAXLEN); //does not null terminate
strcpy(pathbuf,inputbuf); //requires null terminated input
...
```

The code above will behave correctly if the data read from cfgfile is null terminated on disk as expected. But if an attacker is able to modify this input so that it does not contain the expected NULL character, the call to strcpy() will continue copying from memory until it encounters an arbitrary NULL character. This will likely overflow the destination buffer and, if the attacker can control the contents of memory immediately following inputbuf, can leave the application susceptible to a buffer overflow attack.

Example 2:

In the following code, readlink() expands the name of a symbolic link stored in pathname and puts the absolute path into buf. The length of the resulting value is then calculated using strlen().

Example Language: C *(Bad)*

```
char buf[MAXPATH];
...
readlink(pathname, buf, MAXPATH);
int length = strlen(buf);
...
```

The code above will not always behave correctly as readlink() does not append a NULL byte to buf. Readlink() will stop copying characters once the maximum size of buf has been reached to avoid overflowing the buffer, this will leave the value buf not NULL terminated. In this situation, strlen() will continue traversing memory until it encounters an arbitrary NULL character further on down the stack, resulting in a length value that is much larger than the size of string. Readlink() does return the number of bytes copied, but when this return value is the same as stated buf size (in this case MAXPATH), it is impossible to know whether the pathname is precisely that many bytes long, or whether readlink() has truncated the name to avoid overrunning the buffer. In testing, vulnerabilities like this one might not be caught because the unused contents of buf and the memory immediately following it may be NULL, thereby causing strlen() to appear as if it is behaving correctly.

Example 3:

While the following example is not exploitable, it provides a good example of how nulls can be omitted or misplaced, even when "safe" functions are used:

Example Language: C *(Bad)*

```
#include <stdio.h>
#include <string.h>
int main() {
  char longString[] = "String signifying nothing";
  char shortString[16];
  strncpy(shortString, longString, 16);
  printf("The last character in shortString is: %c (%1$x)\n", shortString[15]);
  return (0);
}
```

The above code gives the following output: "The last character in shortString is: n (6e)". So, the shortString array does not end in a NULL character, even though the "safe" string function strncpy() was used. The reason is that strncpy() does not impliciitly add a NULL character at the end of the string when the source is equal in length or longer than the provided size.

Observed Examples

Reference	Description
CVE-2000-0312	Attacker does not null-terminate argv[] when invoking another program. *https://www.cve.org/CVERecord?id=CVE-2000-0312*
CVE-2003-0777	Interrupted step causes resultant lack of null termination. *https://www.cve.org/CVERecord?id=CVE-2003-0777*
CVE-2004-1072	Fault causes resultant lack of null termination, leading to buffer expansion. *https://www.cve.org/CVERecord?id=CVE-2004-1072*
CVE-2001-1389	Multiple vulnerabilities related to improper null termination. *https://www.cve.org/CVERecord?id=CVE-2001-1389*
CVE-2003-0143	Product does not null terminate a message buffer after snprintf-like call, leading to overflow. *https://www.cve.org/CVERecord?id=CVE-2003-0143*
CVE-2009-2523	Chain: product does not handle when an input string is not NULL terminated (CWE-170), leading to buffer over-read (CWE-125) or heap-based buffer overflow (CWE-122). *https://www.cve.org/CVERecord?id=CVE-2009-2523*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	730	OWASP Top Ten 2004 Category A9 - Denial of Service	711	2339
MemberOf	C	741	CERT C Secure Coding Standard (2008) Chapter 8 - Characters and Strings (STR)	734	2344
MemberOf	C	748	CERT C Secure Coding Standard (2008) Appendix - POSIX (POS)	734	2351
MemberOf	C	875	CERT C++ Secure Coding Section 07 - Characters and Strings (STR)	868	2376
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	973	SFP Secondary Cluster: Improper NULL Termination	888	2406
MemberOf	C	1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	1154	2458
MemberOf	C	1171	SEI CERT C Coding Standard - Guidelines 50. POSIX (POS)	1154	2463
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

Factors: this is usually resultant from other weaknesses such as off-by-one errors, but it can be primary to boundary condition violations such as buffer overflows. In buffer overflows, it can act as an expander for assumed-immutable data.

Relationship

Overlaps missing input terminator.

Applicable Platform

Conceptually, this does not just apply to the C language; any language or representation that involves a terminator could have this type of problem.

Maintenance

As currently described, this entry is more like a category than a weakness.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Improper Null Termination
7 Pernicious Kingdoms			String Termination Error
CLASP			Miscalculated null termination
OWASP Top Ten 2004	A9	CWE More Specific	Denial of Service
CERT C Secure Coding	POS30-C	CWE More Abstract	Use the readlink() function properly
CERT C Secure Coding	STR03-C		Do not inadvertently truncate a null-terminated byte string
CERT C Secure Coding	STR32-C	Exact	Do not pass a non-null-terminated character sequence to a library function that expects a string
Software Fault Patterns	SFP11		Improper Null Termination

CWE-172: Encoding Error

Weakness ID : 172
Structure : Simple
Abstraction : Class

Description

The product does not properly encode or decode the data, resulting in unexpected values.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	707	Improper Neutralization	1546
ParentOf	V	173	Improper Handling of Alternate Encoding	435
ParentOf	V	174	Double Decoding of the Same Data	437
ParentOf	V	175	Improper Handling of Mixed Encoding	439
ParentOf	V	176	Improper Handling of Unicode Encoding	440
ParentOf	V	177	Improper Handling of URL Encoding (Hex Encoding)	442
CanPrecede	B	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	33
CanPrecede	B	41	Improper Resolution of Path Equivalence	86

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Phase: 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.

Observed Examples

Reference	Description
CVE-2004-1315	Forum software improperly URL decodes the highlight parameter when extracting text to highlight, which allows remote attackers to execute arbitrary PHP code by double-encoding the highlight value so that special characters are inserted into the result. *https://www.cve.org/CVERecord?id=CVE-2004-1315*
CVE-2004-1939	XSS protection mechanism attempts to remove "/" that could be used to close tags, but it can be bypassed using double encoded slashes (%252F) *https://www.cve.org/CVERecord?id=CVE-2004-1939*
CVE-2001-0709	Server allows a remote attacker to obtain source code of ASP files via a URL encoded with Unicode. *https://www.cve.org/CVERecord?id=CVE-2001-0709*
CVE-2005-2256	Hex-encoded path traversal variants - "%2e%2e", "%2e%2e%2f", "%5c%2e%2e" *https://www.cve.org/CVERecord?id=CVE-2005-2256*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Relationship

Partially overlaps path traversal and equivalence weaknesses.

Maintenance

This is more like a category than a weakness.

Maintenance

Many other types of encodings should be listed in this category.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Encoding Error

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
52	Embedding NULL Bytes
53	Postfix, Null Terminate, and Backslash
64	Using Slashes and URL Encoding Combined to Bypass Validation Logic
71	Using Unicode Encoding to Bypass Validation Logic
72	URL Encoding
78	Using Escaped Slashes in Alternate Encoding
80	Using UTF-8 Encoding to Bypass Validation Logic
120	Double Encoding
267	Leverage Alternate Encoding

CWE-173: Improper Handling of Alternate Encoding

Weakness ID : 173
Structure : Simple
Abstraction : Variant

Description

The product does not properly handle when an input uses an alternate encoding that is valid for the control sphere to which the input is being sent.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	172	Encoding Error	433
CanPrecede	B	289	Authentication Bypass by Alternate Name	703

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component.

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Alternate Encoding

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
4	Using Alternative IP Address Encodings
52	Embedding NULL Bytes
53	Postfix, Null Terminate, and Backslash
64	Using Slashes and URL Encoding Combined to Bypass Validation Logic
71	Using Unicode Encoding to Bypass Validation Logic
72	URL Encoding
78	Using Escaped Slashes in Alternate Encoding
79	Using Slashes in Alternate Encoding
80	Using UTF-8 Encoding to Bypass Validation Logic
120	Double Encoding

CAPEC-ID	Attack Pattern Name
267	Leverage Alternate Encoding

CWE-174: Double Decoding of the Same Data

Weakness ID : 174
Structure : Simple
Abstraction : Variant

Description

The product decodes the same input twice, which can limit the effectiveness of any protection mechanism that occurs in between the decoding operations.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	172	Encoding Error	433
ChildOf	●	675	Multiple Operations on Resource in Single-Operation Context	1487

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control Confidentiality Availability Integrity Other	Bypass Protection Mechanism Execute Unauthorized Code or Commands Varies by Context	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component.

Phase: 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.

Observed Examples

Reference	Description
CVE-2004-1315	Forum software improperly URL decodes the highlight parameter when extracting text to highlight, which allows remote attackers to execute arbitrary PHP code by double-encoding the highlight value so that special characters are inserted into the result. *https://www.cve.org/CVERecord?id=CVE-2004-1315*
CVE-2004-1939	XSS protection mechanism attempts to remove "/" that could be used to close tags, but it can be bypassed using double encoded slashes (%252F) *https://www.cve.org/CVERecord?id=CVE-2004-1939*
CVE-2001-0333	Directory traversal using double encoding. *https://www.cve.org/CVERecord?id=CVE-2001-0333*
CVE-2004-1938	"%2527" (double-encoded single quote) used in SQL injection. *https://www.cve.org/CVERecord?id=CVE-2004-1938*
CVE-2005-1945	Double hex-encoded data. *https://www.cve.org/CVERecord?id=CVE-2005-1945*
CVE-2005-0054	Browser executes HTML at higher privileges via URL with hostnames that are double hex encoded, which are decoded twice to generate a malicious hostname. *https://www.cve.org/CVERecord?id=CVE-2005-0054*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Notes

Research Gap

Probably under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Double Encoding

CWE-175: Improper Handling of Mixed Encoding

Weakness ID : 175
Structure : Simple
Abstraction : Variant

Description

The product does not properly handle when the same input uses several different (mixed) encodings.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	172	Encoding Error	433

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: Implementation

Strategy = Output Encoding

Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component.

Phase: 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.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Mixed Encoding

CWE-176: Improper Handling of Unicode Encoding

Weakness ID : 176
Structure : Simple
Abstraction : Variant

Description

The product does not properly handle when an input contains Unicode encoding.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	172	Encoding Error	433

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: 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.

Demonstrative Examples

Example 1:

Windows provides the MultiByteToWideChar(), WideCharToMultiByte(), UnicodeToBytes(), and BytesToUnicode() functions to convert between arbitrary multibyte (usually ANSI) character strings and Unicode (wide character) strings. The size arguments to these functions are specified in different units, (one in bytes, the other in characters) making their use prone to error.

In a multibyte character string, each character occupies a varying number of bytes, and therefore the size of such strings is most easily specified as a total number of bytes. In Unicode, however, characters are always a fixed size, and string lengths are typically given by the number of characters they contain. Mistakenly specifying the wrong units in a size argument can lead to a buffer overflow.

The following function takes a username specified as a multibyte string and a pointer to a structure for user information and populates the structure with information about the specified user. Since Windows authentication uses Unicode for usernames, the username argument is first converted from a multibyte string to a Unicode string.

Example Language: C *(Bad)*

```
void getUserInfo(char *username, struct _USER_INFO_2 info){
  WCHAR unicodeUser[UNLEN+1];
  MultiByteToWideChar(CP_ACP, 0, username, -1, unicodeUser, sizeof(unicodeUser));
```

```
    NetUserGetInfo(NULL, unicodeUser, 2, (LPBYTE *)&info);
}
```

This function incorrectly passes the size of unicodeUser in bytes instead of characters. The call to MultiByteToWideChar() can therefore write up to (UNLEN+1)*sizeof(WCHAR) wide characters, or (UNLEN+1)*sizeof(WCHAR)*sizeof(WCHAR) bytes, to the unicodeUser array, which has only (UNLEN+1)*sizeof(WCHAR) bytes allocated.

If the username string contains more than UNLEN characters, the call to MultiByteToWideChar() will overflow the buffer unicodeUser.

Observed Examples

Reference	Description
CVE-2000-0884	Server allows remote attackers to read documents outside of the web root, and possibly execute arbitrary commands, via malformed URLs that contain Unicode encoded characters. *https://www.cve.org/CVERecord?id=CVE-2000-0884*
CVE-2001-0709	Server allows a remote attacker to obtain source code of ASP files via a URL encoded with Unicode. *https://www.cve.org/CVERecord?id=CVE-2001-0709*
CVE-2001-0669	Overlaps interaction error. *https://www.cve.org/CVERecord?id=CVE-2001-0669*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Unicode Encoding
CERT C Secure Coding	MSC10-C		Character Encoding - UTF8 Related Issues

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
71	Using Unicode Encoding to Bypass Validation Logic

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-177: Improper Handling of URL Encoding (Hex Encoding)

Weakness ID : 177
Structure : Simple
Abstraction : Variant

Description

The product does not properly handle when all or part of an input has been URL encoded.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	●	172	Encoding Error	433

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: 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.

Observed Examples

Reference	Description
CVE-2000-0900	Hex-encoded path traversal variants - "%2e%2e", "%2e%2e%2f", "%5c%2e%2e" *https://www.cve.org/CVERecord?id=CVE-2000-0900*

Reference	Description
CVE-2005-2256	Hex-encoded path traversal variants - "%2e%2e", "%2e%2e%2f", "%5c%2e %2e" *https://www.cve.org/CVERecord?id=CVE-2005-2256*
CVE-2004-2121	Hex-encoded path traversal variants - "%2e%2e", "%2e%2e%2f", "%5c%2e %2e" *https://www.cve.org/CVERecord?id=CVE-2004-2121*
CVE-2004-0280	"%20" (encoded space) *https://www.cve.org/CVERecord?id=CVE-2004-0280*
CVE-2003-0424	"%20" (encoded space) *https://www.cve.org/CVERecord?id=CVE-2003-0424*
CVE-2001-0693	"%20" (encoded space) *https://www.cve.org/CVERecord?id=CVE-2001-0693*
CVE-2001-0778	"%20" (encoded space) *https://www.cve.org/CVERecord?id=CVE-2001-0778*
CVE-2002-1831	Crash via hex-encoded space "%20". *https://www.cve.org/CVERecord?id=CVE-2002-1831*
CVE-2000-0671	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2000-0671*
CVE-2004-0189	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2004-0189*
CVE-2002-1291	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2002-1291*
CVE-2002-1031	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2002-1031*
CVE-2001-1140	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2001-1140*
CVE-2004-0760	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2004-0760*
CVE-2002-1025	"%00" (encoded null) *https://www.cve.org/CVERecord?id=CVE-2002-1025*
CVE-2002-1213	"%2f" (encoded slash) *https://www.cve.org/CVERecord?id=CVE-2002-1213*
CVE-2004-0072	"%5c" (encoded backslash) and "%2e" (encoded dot) sequences *https://www.cve.org/CVERecord?id=CVE-2004-0072*
CVE-2004-0847	"%5c" (encoded backslash) *https://www.cve.org/CVERecord?id=CVE-2004-0847*
CVE-2002-1575	"%0a" (overlaps CRLF) *https://www.cve.org/CVERecord?id=CVE-2002-1575*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			URL Encoding (Hex Encoding)

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
64	Using Slashes and URL Encoding Combined to Bypass Validation Logic
72	URL Encoding
120	Double Encoding
468	Generic Cross-Browser Cross-Domain Theft

CWE-178: Improper Handling of Case Sensitivity

Weakness ID : 178
Structure : Simple
Abstraction : Base

Description

The product does not properly account for differences in case sensitivity when accessing or determining the properties of a resource, leading to inconsistent results.

Extended Description

Improperly handled case sensitive data can lead to several possible consequences, including:

- case-insensitive passwords reducing the size of the key space, making brute force attacks easier
- bypassing filters or access controls using alternate names
- multiple interpretation errors using alternate names.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	706	Use of Incorrectly-Resolved Name or Reference	1544
PeerOf	B	1289	Improper Validation of Unsafe Equivalence in Input	2141
CanPrecede	B	289	Authentication Bypass by Alternate Name	703
CanPrecede	V	433	Unparsed Raw Web Content Delivery	1046

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	706	Use of Incorrectly-Resolved Name or Reference	1544

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

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: 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.

Demonstrative Examples

Example 1:

In the following example, an XSS neutralization method intends to replace script tags in user-supplied input with a safe equivalent:

Example Language: Java *(Bad)*

```
public String preventXSS(String input, String mask) {
  return input.replaceAll("script", mask);
}
```

The code only works when the "script" tag is in all lower-case, forming an incomplete denylist (CWE-184). Equivalent tags such as "SCRIPT" or "ScRiPt" will not be neutralized by this method, allowing an XSS attack.

Observed Examples

Reference	Description
CVE-2000-0499	Application server allows attackers to bypass execution of a jsp page and read the source code using an upper case JSP extension in the request. *https://www.cve.org/CVERecord?id=CVE-2000-0499*
CVE-2000-0497	The server is case sensitive, so filetype handlers treat .jsp and .JSP as different extensions. JSP source code may be read because .JSP defaults to the filetype "text". *https://www.cve.org/CVERecord?id=CVE-2000-0497*
CVE-2000-0498	The server is case sensitive, so filetype handlers treat .jsp and .JSP as different extensions. JSP source code may be read because .JSP defaults to the filetype "text".

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2000-0498
CVE-2001-0766	A URL that contains some characters whose case is not matched by the server's filters may bypass access restrictions because the case-insensitive file system will then handle the request after it bypasses the case sensitive filter. *https://www.cve.org/CVERecord?id=CVE-2001-0766*
CVE-2001-0795	Server allows remote attackers to obtain source code of CGI scripts via URLs that contain MS-DOS conventions such as (1) upper case letters or (2) 8.3 file names. *https://www.cve.org/CVERecord?id=CVE-2001-0795*
CVE-2001-1238	Task Manager does not allow local users to end processes with uppercase letters named (1) winlogon.exe, (2) csrss.exe, (3) smss.exe and (4) services.exe via the Process tab which could allow local users to install Trojan horses that cannot be stopped. *https://www.cve.org/CVERecord?id=CVE-2001-1238*
CVE-2003-0411	chain: Code was ported from a case-sensitive Unix platform to a case-insensitive Windows platform where filetype handlers treat .jsp and .JSP as different extensions. JSP source code may be read because .JSP defaults to the filetype "text". *https://www.cve.org/CVERecord?id=CVE-2003-0411*
CVE-2002-0485	Leads to interpretation error *https://www.cve.org/CVERecord?id=CVE-2002-0485*
CVE-1999-0239	Directories may be listed because lower case web requests are not properly handled by the server. *https://www.cve.org/CVERecord?id=CVE-1999-0239*
CVE-2005-0269	File extension check in forum software only verifies extensions that contain all lowercase letters, which allows remote attackers to upload arbitrary files via file extensions that include uppercase letters. *https://www.cve.org/CVERecord?id=CVE-2005-0269*
CVE-2004-1083	Web server restricts access to files in a case sensitive manner, but the filesystem accesses files in a case insensitive manner, which allows remote attackers to read privileged files using alternate capitalization. *https://www.cve.org/CVERecord?id=CVE-2004-1083*
CVE-2002-2119	Case insensitive passwords lead to search space reduction. *https://www.cve.org/CVERecord?id=CVE-2002-2119*
CVE-2004-2214	HTTP server allows bypass of access restrictions using URIs with mixed case. *https://www.cve.org/CVERecord?id=CVE-2004-2214*
CVE-2004-2154	Mixed upper/lowercase allows bypass of ACLs. *https://www.cve.org/CVERecord?id=CVE-2004-2154*
CVE-2005-4509	Bypass malicious script detection by using tokens that aren't case sensitive. *https://www.cve.org/CVERecord?id=CVE-2005-4509*
CVE-2002-1820	Mixed case problem allows "admin" to have "Admin" rights (alternate name property). *https://www.cve.org/CVERecord?id=CVE-2002-1820*
CVE-2007-3365	Chain: uppercase file extensions causes web server to return script source code instead of executing the script. *https://www.cve.org/CVERecord?id=CVE-2007-3365*
CVE-2021-39155	Chain: A microservice integration and management platform compares the hostname in the HTTP Host header in a case-sensitive way (CWE-178, CWE-1289), allowing bypass of the authorization policy (CWE-863) using a hostname with mixed case or other variations. *https://www.cve.org/CVERecord?id=CVE-2021-39155*

Functional Areas

- File Processing

Affected Resources

- File or Directory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Research Gap

These are probably under-studied in Windows and Mac environments, where file names are case-insensitive and thus are subject to equivalence manipulations involving case.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Case Sensitivity (lowercase, uppercase, mixed case)

CWE-179: Incorrect Behavior Order: Early Validation

Weakness ID : 179
Structure : Simple
Abstraction : Base

Description

The product validates input before applying protection mechanisms that modify the input, which could allow an attacker to bypass the validation via dangerous inputs that only arise after the modification.

Extended Description

Product needs to validate data at the proper time, after data has been canonicalized and cleansed. Early validation is susceptible to various manipulations that result in dangerous inputs that are produced by canonicalization and cleansing.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20
ChildOf	C	696	Incorrect Behavior Order	1527
ParentOf	V	180	Incorrect Behavior Order: Validate Before Canonicalize	451
ParentOf	V	181	Incorrect Behavior Order: Validate Before Filter	453

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

Nature	Type	ID	Name	Page
MemberOf	C	1215	Data Validation Issues	2478
MemberOf	C	438	Behavioral Problems	2326

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control Integrity	Bypass Protection Mechanism Execute Unauthorized Code or Commands *An attacker could include dangerous input that bypasses validation protection mechanisms which can be used to launch various attacks including injection attacks, execute arbitrary code or cause other unintended behavior.*	

Potential Mitigations

Phase: 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.

Demonstrative Examples

Example 1:

The following code attempts to validate a given input path by checking it against an allowlist and then return the canonical path. In this specific case, the path is considered valid if it starts with the string "/safe_dir/".

Example Language: Java *(Bad)*

```
String path = getInputPath();
if (path.startsWith("/safe_dir/"))
{
  File f = new File(path);
  return f.getCanonicalPath();
}
```

The problem with the above code is that the validation step occurs before canonicalization occurs. An attacker could provide an input path of "/safe_dir/../" that would pass the validation step. However, the canonicalization process sees the double dot as a traversal to the parent directory and hence when canonicized the path would become just "/".

To avoid this problem, validation should occur after canonicalization takes place. In this case canonicalization occurs during the initialization of the File object. The code below fixes the issue.

Example Language: Java *(Good)*

```
String path = getInputPath();
File f = new File(path);
if (f.getCanonicalPath().startsWith("/safe_dir/"))
{
  return f.getCanonicalPath();
}
```

Example 2:

This script creates a subdirectory within a user directory and sets the user as the owner.

Example Language: PHP *(Bad)*

```
function createDir($userName,$dirName){
  $userDir = '/users/'. $userName;
  if(strpos($dirName,'..') !== false){
    echo 'Directory name contains invalid sequence';
    return;
  }
  //filter out '~' because other scripts identify user directories by this prefix
  $dirName = str_replace('~','',$dirName);
  $newDir = $userDir . $dirName;
  mkdir($newDir, 0700);
  chown($newDir,$userName);
}
```

While the script attempts to screen for '..' sequences, an attacker can submit a directory path including ".~.", which will then become ".." after the filtering step. This allows a Path Traversal (CWE-21) attack to occur.

Observed Examples

Reference	Description
CVE-2002-0433	Product allows remote attackers to view restricted files via an HTTP request containing a "*" (wildcard or asterisk) character. *https://www.cve.org/CVERecord?id=CVE-2002-0433*
CVE-2003-0332	Product modifies the first two letters of a filename extension after performing a security check, which allows remote attackers to bypass authentication via a filename with a .ats extension instead of a .hts extension. *https://www.cve.org/CVERecord?id=CVE-2003-0332*
CVE-2002-0802	Database consumes an extra character when processing a character that cannot be converted, which could remove an escape character from the query and make the application subject to SQL injection attacks. *https://www.cve.org/CVERecord?id=CVE-2002-0802*
CVE-2000-0191	Overlaps "fakechild/../realchild" *https://www.cve.org/CVERecord?id=CVE-2000-0191*
CVE-2004-2363	Product checks URI for "<" and other literal characters, but does it before hex decoding the URI, so "%3E" and other sequences are allowed. *https://www.cve.org/CVERecord?id=CVE-2004-2363*
CVE-2002-0934	Directory traversal vulnerability allows remote attackers to read or modify arbitrary files via invalid characters between two . (dot) characters, which are filtered and result in a ".." sequence. *https://www.cve.org/CVERecord?id=CVE-2002-0934*
CVE-2003-0282	Directory traversal vulnerability allows attackers to overwrite arbitrary files via invalid characters between two . (dot) characters, which are filtered and result in a ".." sequence. *https://www.cve.org/CVERecord?id=CVE-2003-0282*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416

Nature	Type	ID	Name	V	Page
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Notes

Research Gap

These errors are mostly reported in path traversal vulnerabilities, but the concept applies whenever validation occurs.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Early Validation Errors

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
43	Exploiting Multiple Input Interpretation Layers
71	Using Unicode Encoding to Bypass Validation Logic

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-180: Incorrect Behavior Order: Validate Before Canonicalize

Weakness ID : 180
Structure : Simple
Abstraction : Variant

Description

The product validates input before it is canonicalized, which prevents the product from detecting data that becomes invalid after the canonicalization step.

Extended Description

This can be used by an attacker to bypass the validation and launch attacks that expose weaknesses that would otherwise be prevented, such as injection.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	179	Incorrect Behavior Order: Early Validation	448

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: 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.

Demonstrative Examples

Example 1:

The following code attempts to validate a given input path by checking it against an allowlist and then return the canonical path. In this specific case, the path is considered valid if it starts with the string "/safe_dir/".

Example Language: Java *(Bad)*

```
String path = getInputPath();
if (path.startsWith("/safe_dir/"))
{
  File f = new File(path);
  return f.getCanonicalPath();
}
```

The problem with the above code is that the validation step occurs before canonicalization occurs. An attacker could provide an input path of "/safe_dir/../" that would pass the validation step. However, the canonicalization process sees the double dot as a traversal to the parent directory and hence when canonicized the path would become just "/".

To avoid this problem, validation should occur after canonicalization takes place. In this case canonicalization occurs during the initialization of the File object. The code below fixes the issue.

Example Language: Java *(Good)*

```
String path = getInputPath();
File f = new File(path);
if (f.getCanonicalPath().startsWith("/safe_dir/"))
{
  return f.getCanonicalPath();
}
```

Observed Examples

Reference	Description
CVE-2002-0433	Product allows remote attackers to view restricted files via an HTTP request containing a "*" (wildcard or asterisk) character. *https://www.cve.org/CVERecord?id=CVE-2002-0433*
CVE-2003-0332	Product modifies the first two letters of a filename extension after performing a security check, which allows remote attackers to bypass authentication via a filename with a .ats extension instead of a .hts extension. *https://www.cve.org/CVERecord?id=CVE-2003-0332*
CVE-2002-0802	Database consumes an extra character when processing a character that cannot be converted, which could remove an escape character from the query and make the application subject to SQL injection attacks. *https://www.cve.org/CVERecord?id=CVE-2002-0802*
CVE-2000-0191	Overlaps "fakechild/../realchild" *https://www.cve.org/CVERecord?id=CVE-2000-0191*
CVE-2004-2363	Product checks URI for "<" and other literal characters, but does it before hex decoding the URI, so "%3E" and other sequences are allowed. *https://www.cve.org/CVERecord?id=CVE-2004-2363*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	845	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 2 - Input Validation and Data Sanitization (IDS)	844	2362
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1147	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 13. Input Output (FIO)	1133	2450
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Notes

Relationship

This overlaps other categories.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Validate-Before-Canonicalize
OWASP Top Ten 2004	A1	CWE More Specific	Unvalidated Input
The CERT Oracle Secure Coding Standard for Java (2011)	IDS01-J	Exact	Normalize strings before validating them
SEI CERT Oracle Coding Standard for Java	IDS01-J	Exact	Normalize strings before validating them

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
71	Using Unicode Encoding to Bypass Validation Logic
78	Using Escaped Slashes in Alternate Encoding
79	Using Slashes in Alternate Encoding
80	Using UTF-8 Encoding to Bypass Validation Logic
267	Leverage Alternate Encoding

CWE-181: Incorrect Behavior Order: Validate Before Filter

Weakness ID : 181
Structure : Simple
Abstraction : Variant

Description

The product validates data before it has been filtered, which prevents the product from detecting data that becomes invalid after the filtering step.

Extended Description

This can be used by an attacker to bypass the validation and launch attacks that expose weaknesses that would otherwise be prevented, such as injection.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	179	Incorrect Behavior Order: Early Validation	448

Applicable Platforms

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

Alternate Terms

Validate-before-cleanse :

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Implementation

Phase: Architecture and Design

Inputs should be decoded and canonicalized to the application's current internal representation before being filtered.

Demonstrative Examples

Example 1:

This script creates a subdirectory within a user directory and sets the user as the owner.

Example Language: PHP *(Bad)*

```
function createDir($userName,$dirName){
  $userDir = '/users/'. $userName;
  if(strpos($dirName,'..') !== false){
    echo 'Directory name contains invalid sequence';
    return;
  }
  //filter out '~' because other scripts identify user directories by this prefix
  $dirName = str_replace('~','',$dirName);
  $newDir = $userDir . $dirName;
  mkdir($newDir, 0700);
  chown($newDir,$userName);
}
```

While the script attempts to screen for '..' sequences, an attacker can submit a directory path including ".~.", which will then become ".." after the filtering step. This allows a Path Traversal (CWE-21) attack to occur.

Observed Examples

Reference	Description
CVE-2002-0934	Directory traversal vulnerability allows remote attackers to read or modify arbitrary files via invalid characters between two . (dot) characters, which are filtered and result in a ".." sequence.

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2002-0934
CVE-2003-0282	Directory traversal vulnerability allows attackers to overwrite arbitrary files via invalid characters between two . (dot) characters, which are filtered and result in a ".." sequence. *https://www.cve.org/CVERecord?id=CVE-2003-0282*

Functional Areas

- Protection Mechanism

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Notes

Research Gap

This category is probably under-studied.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Validate-Before-Filter
OWASP Top Ten 2004	A1	CWE More Specific	Unvalidated Input

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
43	Exploiting Multiple Input Interpretation Layers
78	Using Escaped Slashes in Alternate Encoding
79	Using Slashes in Alternate Encoding
80	Using UTF-8 Encoding to Bypass Validation Logic
120	Double Encoding
267	Leverage Alternate Encoding

CWE-182: Collapse of Data into Unsafe Value

Weakness ID : 182
Structure : Simple
Abstraction : Base

Description

The product filters data in a way that causes it to be reduced or "collapsed" into an unsafe value that violates an expected security property.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	693	Protection Mechanism Failure	1520
CanFollow	C	185	Incorrect Regular Expression	463
CanPrecede	V	33	Path Traversal: '....' (Multiple Dot)	69
CanPrecede	V	34	Path Traversal: '....//'	71
CanPrecede	V	35	Path Traversal: '.../...//'	73

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

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

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

Potential Mitigations

Phase: Architecture and Design

Strategy = Input Validation

Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.

Phase: 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.

Phase: 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.

Canonicalize the name to match that of the file system's representation of the name. This can sometimes be achieved with an available API (e.g. in Win32 the GetFullPathName function).

Observed Examples

Reference	Description
CVE-2004-0815	"/.////" in pathname collapses to absolute path. *https://www.cve.org/CVERecord?id=CVE-2004-0815*
CVE-2005-3123	"/.//..//////././" is collapsed into "/.././" after ".." and "//" sequences are removed. *https://www.cve.org/CVERecord?id=CVE-2005-3123*
CVE-2002-0325	".../...//" collapsed to "..." due to removal of "./" in web server. *https://www.cve.org/CVERecord?id=CVE-2002-0325*
CVE-2002-0784	chain: HTTP server protects against ".." but allows "." variants such as "////./../.../". If the server removes "/.." sequences, the result would collapse into an unsafe value "////../" (CWE-182). *https://www.cve.org/CVERecord?id=CVE-2002-0784*
CVE-2005-2169	MFV. Regular expression intended to protect against directory traversal reduces ".../...//" to "../". *https://www.cve.org/CVERecord?id=CVE-2005-2169*
CVE-2001-1157	XSS protection mechanism strips a <script> sequence that is nested in another <script> sequence. *https://www.cve.org/CVERecord?id=CVE-2001-1157*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	845	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 2 - Input Validation and Data Sanitization (IDS)	844	2362
MemberOf	C	992	SFP Secondary Cluster: Faulty Input Transformation	888	2416
MemberOf	C	1134	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 00. Input Validation and Data Sanitization (IDS)	1133	2444
MemberOf	C	1413	Comprehensive Categorization: Protection Mechanism Failure	1400	2542

Notes

Relationship

Overlaps regular expressions, although an implementation might not necessarily use regexp's.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Collapse of Data into Unsafe Value
The CERT Oracle Secure Coding Standard for Java (2011)	IDS11-J		Eliminate noncharacter code points before validation

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-183: Permissive List of Allowed Inputs

Weakness ID : 183
Structure : Simple
Abstraction : Base

Description

The product implements a protection mechanism that relies on a list of inputs (or properties of inputs) that are explicitly allowed by policy because the inputs are assumed to be safe, but the list is too permissive - that is, it allows an input that is unsafe, leading to resultant weaknesses.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	697	Incorrect Comparison	1530
ParentOf	V	942	Permissive Cross-domain Policy with Untrusted Domains	1847
PeerOf	B	625	Permissive Regular Expression	1392
PeerOf	V	627	Dynamic Variable Evaluation	1396
CanPrecede	B	434	Unrestricted Upload of File with Dangerous Type	1048

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

Nature	Type	ID	Name	Page
MemberOf	C	1215	Data Validation Issues	2478

Weakness Ordinalities

Primary :

Applicable Platforms

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

Alternate Terms

Allowlist / Allow List : This is used by CWE and CAPEC instead of other commonly-used terms. Its counterpart is denylist.

Safelist / Safe List : This is often used by security tools such as firewalls, email or web gateways, proxies, etc.

Whitelist / White List : This term is frequently used, but usage has been declining as organizations have started to adopt other terms.

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

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

Observed Examples

Reference	Description
CVE-2019-12799	chain: bypass of untrusted deserialization issue (CWE-502) by using an assumed-trusted class (CWE-183) *https://www.cve.org/CVERecord?id=CVE-2019-12799*
CVE-2019-10458	sandbox bypass using a method that is on an allowlist *https://www.cve.org/CVERecord?id=CVE-2019-10458*
CVE-2017-1000095	sandbox bypass using unsafe methods that are on an allowlist *https://www.cve.org/CVERecord?id=CVE-2017-1000095*
CVE-2019-10458	CI/CD pipeline feature has unsafe elements in allowlist, allowing bypass of script restrictions *https://www.cve.org/CVERecord?id=CVE-2019-10458*
CVE-2017-1000095	Default allowlist includes unsafe methods, allowing bypass of sandbox *https://www.cve.org/CVERecord?id=CVE-2017-1000095*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Permissive Whitelist

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
43	Exploiting Multiple Input Interpretation Layers
71	Using Unicode Encoding to Bypass Validation Logic
120	Double Encoding

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-184: Incomplete List of Disallowed Inputs

Weakness ID : 184
Structure : Simple

Abstraction : Base

Description

The product implements a protection mechanism that relies on a list of inputs (or properties of inputs) that are not allowed by policy or otherwise require other action to neutralize before additional processing takes place, but the list is incomplete, leading to resultant weaknesses.

Extended Description

Developers often try to protect their products against malicious input by performing tests against inputs that are known to be bad, such as special characters that can invoke new commands. However, such lists often only account for the most well-known bad inputs. Attackers may be able to find other malicious inputs that were not expected by the developer, allowing them to bypass the intended protection mechanism.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	1023	Incomplete Comparison with Missing Factors	1865
ChildOf	P	693	Protection Mechanism Failure	1520
ParentOf	⚯	692	Incomplete Denylist to Cross-Site Scripting	1519
PeerOf	V	86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages	190
PeerOf	B	625	Permissive Regular Expression	1392
CanPrecede	B	78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	151
CanPrecede	B	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	163
CanPrecede	V	98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')	236
CanPrecede	B	434	Unrestricted Upload of File with Dangerous Type	1048

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

Nature	Type	ID	Name	Page
MemberOf	C	1215	Data Validation Issues	2478

Weakness Ordinalities

Primary :

Applicable Platforms

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

Alternate Terms

Denylist / Deny List : This is used by CWE and CAPEC instead of other commonly-used terms. Its counterpart is allowlist.

Blocklist / Block List : This is often used by security tools such as firewalls, email or web gateways, proxies, etc.

Blacklist / Black List : This term is frequently used, but usage has been declining as organizations have started to adopt other terms.

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Detection Methods

Black Box

Exploitation of a vulnerability with commonly-used manipulations might fail, but minor variations might succeed.

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Do not rely exclusively on detecting disallowed inputs. There are too many variants to encode a character, especially when different environments are used, so there is a high likelihood of missing some variants. Only use detection of disallowed inputs as a mechanism for detecting suspicious activity. Ensure that you are using other protection mechanisms that only identify "good" input - such as lists of allowed inputs - and ensure that you are properly encoding your outputs.

Demonstrative Examples

Example 1:

The following code attempts to stop XSS attacks by removing all occurences of "script" in an input string.

Example Language: Java *(Bad)*

```
public String removeScriptTags(String input, String mask) {
  return input.replaceAll("script", mask);
}
```

Because the code only checks for the lower-case "script" string, it can be easily defeated with upper-case script tags.

Observed Examples

Reference	Description
CVE-2008-2309	product uses a denylist to identify potentially dangerous content, allowing attacker to bypass a warning *https://www.cve.org/CVERecord?id=CVE-2008-2309*
CVE-2005-2782	PHP remote file inclusion in web application that filters "http" and "https" URLs, but not "ftp". *https://www.cve.org/CVERecord?id=CVE-2005-2782*
CVE-2004-0542	Programming language does not filter certain shell metacharacters in Windows environment. *https://www.cve.org/CVERecord?id=CVE-2004-0542*
CVE-2004-0595	XSS filter doesn't filter null characters before looking for dangerous tags, which are ignored by web browsers. MIE and validate-before-cleanse. *https://www.cve.org/CVERecord?id=CVE-2004-0595*
CVE-2005-3287	Web-based mail product doesn't restrict dangerous extensions such as ASPX on a web server, even though others are prohibited. *https://www.cve.org/CVERecord?id=CVE-2005-3287*
CVE-2004-2351	Resultant XSS when only <script> and <style> are checked. *https://www.cve.org/CVERecord?id=CVE-2004-2351*
CVE-2005-2959	Privileged program does not clear sensitive environment variables that are used by bash. Overlaps multiple interpretation error. *https://www.cve.org/CVERecord?id=CVE-2005-2959*

Reference	Description
CVE-2005-1824	SQL injection protection scheme does not quote the "\" special character. *https://www.cve.org/CVERecord?id=CVE-2005-1824*
CVE-2005-2184	Detection of risky filename extensions prevents users from automatically executing .EXE files, but .LNK is accepted, allowing resultant Windows symbolic link. *https://www.cve.org/CVERecord?id=CVE-2005-2184*
CVE-2007-1343	Product uses list of protected variables, but accidentally omits one dangerous variable, allowing external modification *https://www.cve.org/CVERecord?id=CVE-2007-1343*
CVE-2007-5727	Chain: product only removes SCRIPT tags (CWE-184), enabling XSS (CWE-79) *https://www.cve.org/CVERecord?id=CVE-2007-5727*
CVE-2006-4308	Chain: product only checks for use of "javascript:" tag (CWE-184), allowing XSS (CWE-79) using other tags *https://www.cve.org/CVERecord?id=CVE-2006-4308*
CVE-2007-3572	Chain: OS command injection (CWE-78) enabled by using an unexpected character that is not explicitly disallowed (CWE-184) *https://www.cve.org/CVERecord?id=CVE-2007-3572*
CVE-2002-0661	"\" not in list of disallowed values for web server, allowing path traversal attacks when the server is run on Windows and other OSes. *https://www.cve.org/CVERecord?id=CVE-2002-0661*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1413	Comprehensive Categorization: Protection Mechanism Failure	1400	2542

Notes

Relationship

Multiple interpretation errors can indirectly introduce inputs that should be disallowed. For example, a list of dangerous shell metacharacters might not include a metacharacter that only has meaning in one particular shell, not all of them; or a check for XSS manipulations might ignore an unusual construct that is supported by one web browser, but not others.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Incomplete Blacklist

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
3	Using Leading 'Ghost' Character Sequences to Bypass Input Filters
6	Argument Injection
15	Command Delimiters
43	Exploiting Multiple Input Interpretation Layers
71	Using Unicode Encoding to Bypass Validation Logic
73	User-Controlled Filename
85	AJAX Footprinting
120	Double Encoding

CAPEC-ID	Attack Pattern Name
182	Flash Injection

References

[REF-140]Greg Hoglund and Gary McGraw. "Exploiting Software: How to Break Code". 2004 February 7. Addison-Wesley. < https://www.amazon.com/Exploiting-Software-How-Break-Code/dp/0201786958 >.2023-04-07.

[REF-141]Steve Christey. "Blacklist defenses as a breeding ground for vulnerability variants". 2006 February 3. < https://seclists.org/fulldisclosure/2006/Feb/40 >.2023-04-07.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-185: Incorrect Regular Expression

Weakness ID : 185
Structure : Simple
Abstraction : Class

Description

The product specifies a regular expression in a way that causes data to be improperly matched or compared.

Extended Description

When the regular expression is used in protection mechanisms such as filtering or validation, this may allow an attacker to bypass the intended restrictions on the incoming data.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	697	Incorrect Comparison	1530
ParentOf	B	186	Overly Restrictive Regular Expression	466
ParentOf	B	625	Permissive Regular Expression	1392
CanPrecede	B	182	Collapse of Data into Unsafe Value	455
CanPrecede	V	187	Partial String Comparison	467

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Other	Unexpected State Varies by Context *When the regular expression is not correctly specified, data might have a different format or type than the rest of the program expects, producing resultant weaknesses or errors.*	
Access Control	Bypass Protection Mechanism	

Scope	Impact	Likelihood
	In PHP, regular expression checks can sometimes be bypassed with a null byte, leading to any number of weaknesses.	

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

Potential Mitigations

Phase: Architecture and Design

Strategy = Refactoring

Regular expressions can become error prone when defining a complex language even for those experienced in writing grammars. Determine if several smaller regular expressions simplify one large regular expression. Also, subject the regular expression to thorough testing techniques such as equivalence partitioning, boundary value analysis, and robustness. After testing and a reasonable confidence level is achieved, a regular expression may not be foolproof. If an exploit is allowed to slip through, then record the exploit and refactor the regular expression.

Demonstrative Examples

Example 1:

The following code takes phone numbers as input, and uses a regular expression to reject invalid phone numbers.

Example Language: Perl *(Bad)*

```
$phone = GetPhoneNumber();
if ($phone =~ /\d+-\d+/) {
  # looks like it only has hyphens and digits
  system("lookup-phone $phone");
}
else {
  error("malformed number!");
}
```

An attacker could provide an argument such as: "; ls -l ; echo 123-456" This would pass the check, since "123-456" is sufficient to match the "\d+-\d+" portion of the regular expression.

Example 2:

This code uses a regular expression to validate an IP string prior to using it in a call to the "ping" command.

Example Language: Python *(Bad)*

```
import subprocess
import re
def validate_ip_regex(ip: str):
  ip_validator = re.compile(r"((25[0-5]|(2[0-4]|1\d|[1-9]|)\d)\.?\b){4}")
  if ip_validator.match(ip):
    return ip
  else:
    raise ValueError("IP address does not match valid pattern.")
```

```
def run_ping_regex(ip: str):
    validated = validate_ip_regex(ip)
    # The ping command treats zero-prepended IP addresses as octal
    result = subprocess.call(["ping", validated])
    print(result)
```

Since the regular expression does not have anchors (CWE-777), i.e. is unbounded without ^ or $ characters, then prepending a 0 or 0x to the beginning of the IP address will still result in a matched regex pattern. Since the ping command supports octal and hex prepended IP addresses, it will use the unexpectedly valid IP address (CWE-1389). For example, "0x63.63.63.63" would be considered equivalent to "99.63.63.63". As a result, the attacker could potentially ping systems that the attacker cannot reach directly.

Observed Examples

Reference	Description
CVE-2002-2109	Regexp isn't "anchored" to the beginning or end, which allows spoofed values that have trusted values as substrings. *https://www.cve.org/CVERecord?id=CVE-2002-2109*
CVE-2005-1949	Regexp for IP address isn't anchored at the end, allowing appending of shell metacharacters. *https://www.cve.org/CVERecord?id=CVE-2005-1949*
CVE-2001-1072	Bypass access restrictions via multiple leading slash, which causes a regular expression to fail. *https://www.cve.org/CVERecord?id=CVE-2001-1072*
CVE-2000-0115	Local user DoS via invalid regular expressions. *https://www.cve.org/CVERecord?id=CVE-2000-0115*
CVE-2002-1527	chain: Malformed input generates a regular expression error that leads to information exposure. *https://www.cve.org/CVERecord?id=CVE-2002-1527*
CVE-2005-1061	Certain strings are later used in a regexp, leading to a resultant crash. *https://www.cve.org/CVERecord?id=CVE-2005-1061*
CVE-2005-2169	MFV. Regular expression intended to protect against directory traversal reduces ".../...//" to "../". *https://www.cve.org/CVERecord?id=CVE-2005-2169*
CVE-2005-0603	Malformed regexp syntax leads to information exposure in error message. *https://www.cve.org/CVERecord?id=CVE-2005-0603*
CVE-2005-1820	Code injection due to improper quoting of regular expression. *https://www.cve.org/CVERecord?id=CVE-2005-1820*
CVE-2005-3153	Null byte bypasses PHP regexp check. *https://www.cve.org/CVERecord?id=CVE-2005-3153*
CVE-2005-4155	Null byte bypasses PHP regexp check. *https://www.cve.org/CVERecord?id=CVE-2005-4155*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Notes

Relationship

While there is some overlap with allowlist/denylist problems, this entry is intended to deal with incorrectly written regular expressions, regardless of their intended use. Not every regular expression is intended for use as an allowlist or denylist. In addition, allowlists and denylists can be implemented using other mechanisms besides regular expressions.

Research Gap

Regexp errors are likely a primary factor in many MFVs, especially those that require multiple manipulations to exploit. However, they are rarely diagnosed at this level of detail.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Regular Expression Error

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
6	Argument Injection
15	Command Delimiters
79	Using Slashes in Alternate Encoding

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

CWE-186: Overly Restrictive Regular Expression

Weakness ID : 186
Structure : Simple
Abstraction : Base

Description

A regular expression is overly restrictive, which prevents dangerous values from being detected.

Extended Description

This weakness is not about regular expression complexity. Rather, it is about a regular expression that does not match all values that are intended. Consider the use of a regexp to identify acceptable values or to spot unwanted terms. An overly restrictive regexp misses some potentially security-relevant values leading to either false positives *or* false negatives, depending on how the regexp is being used within the code. Consider the expression /[0-8]/ where the intention was /[0-9]/. This expression is not "complex" but the value "9" is not matched when maybe the programmer planned to check for it.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	185	Incorrect Regular Expression	463
CanAlsoBe	Ⓑ	183	Permissive List of Allowed Inputs	458
CanAlsoBe	Ⓑ	184	Incomplete List of Disallowed Inputs	459

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

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Implementation

Regular expressions can become error prone when defining a complex language even for those experienced in writing grammars. Determine if several smaller regular expressions simplify one large regular expression. Also, subject your regular expression to thorough testing techniques such as equivalence partitioning, boundary value analysis, and robustness. After testing and a reasonable confidence level is achieved, a regular expression may not be foolproof. If an exploit is allowed to slip through, then record the exploit and refactor your regular expression.

Observed Examples

Reference	Description
CVE-2005-1604	MIE. ".php.ns" bypasses ".php$" regexp but is still parsed as PHP by Apache. (manipulates an equivalence property under Apache) *https://www.cve.org/CVERecord?id=CVE-2005-1604*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Notes

Relationship

Can overlap allowlist/denylist errors (CWE-183/CWE-184)

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Overly Restrictive Regular Expression

CWE-187: Partial String Comparison

Weakness ID : 187
Structure : Simple
Abstraction : Variant

Description

The product performs a comparison that only examines a portion of a factor before determining whether there is a match, such as a substring, leading to resultant weaknesses.

Extended Description

For example, an attacker might succeed in authentication by providing a small password that matches the associated portion of the larger, correct password.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	1023	Incomplete Comparison with Missing Factors	1865
PeerOf	Ⓑ	625	Permissive Regular Expression	1392
CanFollow	Ⓒ	185	Incorrect Regular Expression	463

Weakness Ordinalities

Primary :

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity Access Control	Alter Execution Logic Bypass Protection Mechanism	

Potential Mitigations

Phase: Testing

Thoroughly test the comparison scheme before deploying code into production. Perform positive testing as well as negative testing.

Demonstrative Examples

Example 1:

This example defines a fixed username and password. The AuthenticateUser() function is intended to accept a username and a password from an untrusted user, and check to ensure that it matches the username and password. If the username and password match, AuthenticateUser() is intended to indicate that authentication succeeded.

Example Language: C *(Bad)*

```
/* Ignore CWE-259 (hard-coded password) and CWE-309 (use of password system for authentication) for this example. */
char *username = "admin";
char *pass = "password";
int AuthenticateUser(char *inUser, char *inPass) {
  if (strncmp(username, inUser, strlen(inUser))) {
    logEvent("Auth failure of username using strlen of inUser");
    return(AUTH_FAIL);
  }
  if (! strncmp(pass, inPass, strlen(inPass))) {
    logEvent("Auth success of password using strlen of inUser");
    return(AUTH_SUCCESS);
  }
  else {
    logEvent("Auth fail of password using sizeof");
    return(AUTH_FAIL);
  }
}
int main (int argc, char **argv) {
  int authResult;
```

```
    if (argc < 3) {
        ExitError("Usage: Provide a username and password");
    }
    authResult = AuthenticateUser(argv[1], argv[2]);
    if (authResult == AUTH_SUCCESS) {
        DoAuthenticatedTask(argv[1]);
    }
    else {
        ExitError("Authentication failed");
    }
}
```

In AuthenticateUser(), the strncmp() call uses the string length of an attacker-provided inPass parameter in order to determine how many characters to check in the password. So, if the attacker only provides a password of length 1, the check will only examine the first byte of the application's password before determining success.

As a result, this partial comparison leads to improper authentication (CWE-287).

Any of these passwords would still cause authentication to succeed for the "admin" user:

Example Language: *(Attack)*

```
p
pa
pas
pass
```

This significantly reduces the search space for an attacker, making brute force attacks more feasible.

The same problem also applies to the username, so values such as "a" and "adm" will succeed for the username.

While this demonstrative example may not seem realistic, see the Observed Examples for CVE entries that effectively reflect this same weakness.

Observed Examples

Reference	Description
CVE-2014-6394	Product does not prevent access to restricted directories due to partial string comparison with a public directory *https://www.cve.org/CVERecord?id=CVE-2014-6394*
CVE-2004-1012	Argument parser of an IMAP server treats a partial command "body[p" as if it is "body.peek", leading to index error and out-of-bounds corruption. *https://www.cve.org/CVERecord?id=CVE-2004-1012*
CVE-2004-0765	Web browser only checks the hostname portion of a certificate when the hostname portion of the URI is not a fully qualified domain name (FQDN), which allows remote attackers to spoof trusted certificates. *https://www.cve.org/CVERecord?id=CVE-2004-0765*
CVE-2002-1374	One-character password by attacker checks only against first character of real password. *https://www.cve.org/CVERecord?id=CVE-2002-1374*
CVE-2000-0979	One-character password by attacker checks only against first character of real password. *https://www.cve.org/CVERecord?id=CVE-2000-0979*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Notes

Relationship

This is conceptually similar to other weaknesses, such as insufficient verification and regular expression errors. It is primary to some weaknesses.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Partial Comparison

CWE-188: Reliance on Data/Memory Layout

Weakness ID : 188
Structure : Simple
Abstraction : Base

Description

The product makes invalid assumptions about how protocol data or memory is organized at a lower level, resulting in unintended program behavior.

Extended Description

When changing platforms or protocol versions, in-memory organization of data may change in unintended ways. For example, some architectures may place local variables A and B right next to each other with A on top; some may place them next to each other with B on top; and others may add some padding to each. The padding size may vary to ensure that each variable is aligned to a proper word size.

In protocol implementations, it is common to calculate an offset relative to another field to pick out a specific piece of data. Exceptional conditions, often involving new protocol versions, may add corner cases that change the data layout in an unusual way. The result can be that an implementation accesses an unintended field in the packet, treating data of one type as data of another type.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	435	Improper Interaction Between Multiple Correctly-Behaving Entities	1055
ChildOf	B	1105	Insufficient Encapsulation of Machine-Dependent Functionality	1945
ParentOf	V	198	Use of Incorrect Byte Ordering	503

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality	Modify Memory Read Memory *Can result in unintended modifications or exposure of sensitive memory.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Potential Mitigations

Phase: Implementation

Phase: Architecture and Design

In flat address space situations, never allow computing memory addresses as offsets from another memory address.

Phase: Architecture and Design

Fully specify protocol layout unambiguously, providing a structured grammar (e.g., a compilable yacc grammar).

Phase: Testing

Testing: Test that the implementation properly handles each case in the protocol grammar.

Demonstrative Examples

Example 1:

In this example function, the memory address of variable b is derived by adding 1 to the address of variable a. This derived address is then used to assign the value 0 to b.

Example Language: C *(Bad)*

```
void example() {
  char a;
  char b;
  *(&a + 1) = 0;
}
```

Here, b may not be one byte past a. It may be one byte in front of a. Or, they may have three bytes between them because they are aligned on 32-bit boundaries.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407

Nature	Type	ID	Name	V	Page
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Reliance on data layout

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-190: Integer Overflow or Wraparound

Weakness ID : 190
Structure : Simple
Abstraction : Base

Description

The product performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

Extended Description

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499
ParentOf	⚯	680	Integer Overflow to Buffer Overflow	1493
PeerOf	B	128	Wrap-around Error	339
PeerOf	B	1339	Insufficient Precision or Accuracy of a Real Number	2242
CanPrecede	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
MemberOf	C	189	Numeric Errors	2312

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	●	20	Improper Input Validation	20

Applicable Platforms

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

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) DoS: Instability *This weakness will generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.*	
Integrity	Modify Memory *If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the wrap around results in other conditions such as buffer overflows, further memory corruption may occur.*	
Confidentiality Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism *This weakness can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.*	

Detection Methods

Automated Static Analysis

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives.

Effectiveness = High

Black Box

Sometimes, evidence of this weakness can be detected using dynamic tools and techniques that interact with the product using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The product's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Effectiveness = Moderate

Without visibility into the code, black box methods may not be able to sufficiently distinguish this weakness from others, requiring follow-up manual methods to diagnose the underlying problem.

Manual Analysis

This weakness can be detected using tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. Specifically, manual static analysis is useful for evaluating the correctness of allocation calculations. This can be useful for detecting overflow conditions (CWE-190) or similar weaknesses that might have serious security impacts on the program.

Effectiveness = High

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

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Bytecode Weakness Analysis - including disassembler + source code weakness analysis Binary Weakness Analysis - including disassembler + source code weakness analysis

Effectiveness = High

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Fuzz Tester Framework-based Fuzzer

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Manual Source Code Review (not inspections)

Effectiveness = SOAR Partial

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

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Requirements

Ensure that all protocols are strictly defined, such that all out-of-bounds behavior can be identified simply, and require strict conformance to the protocol.

Phase: Requirements

Strategy = Language Selection

Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. If possible, choose a language or compiler that performs automatic bounds checking.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Use libraries or frameworks that make it easier to handle numbers without unexpected consequences. Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++). [REF-106]

Phase: Implementation

Strategy = Input Validation

Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range. Use unsigned integers where possible. This makes it easier to perform validation for integer overflows. When signed integers are required, ensure that the range check includes minimum values as well as maximum values.

Phase: Implementation

Understand the programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how the language handles numbers that are too large or too small for its underlying representation. [REF-7] Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation.

Phase: 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.

Phase: Implementation

Strategy = Compilation or Build Hardening

Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.

Demonstrative Examples

Example 1:

The following image processing code allocates a table for images.

Example Language: C *(Bad)*

```
img_t table_ptr; /*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

This code intends to allocate a table of size num_imgs, however as num_imgs grows large, the calculation determining the size of the list will eventually overflow (CWE-190). This will result in a very small list to be allocated instead. If the subsequent code operates on the list as if it were num_imgs long, it may result in many types of out-of-bounds problems (CWE-119).

Example 2:

The following code excerpt from OpenSSH 3.3 demonstrates a classic case of integer overflow:

Example Language: C *(Bad)*

```
nresp = packet_get_int();
if (nresp > 0) {
  response = xmalloc(nresp*sizeof(char*));
  for (i = 0; i < nresp; i++) response[i] = packet_get_string(NULL);
}
```

If nresp has the value 1073741824 and sizeof(char*) has its typical value of 4, then the result of the operation nresp*sizeof(char*) overflows, and the argument to xmalloc() will be 0. Most malloc() implementations will happily allocate a 0-byte buffer, causing the subsequent loop iterations to overflow the heap buffer response.

Example 3:

Integer overflows can be complicated and difficult to detect. The following example is an attempt to show how an integer overflow may lead to undefined looping behavior:

Example Language: C *(Bad)*

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];
while(bytesRec < MAXGET) {
   bytesRec += getFromInput(buf+bytesRec);
}
```

In the above case, it is entirely possible that bytesRec may overflow, continuously creating a lower number than MAXGET and also overwriting the first MAXGET-1 bytes of buf.

Example 4:

In this example the method determineFirstQuarterRevenue is used to determine the first quarter revenue for an accounting/business application. The method retrieves the monthly sales totals for the first three months of the year, calculates the first quarter sales totals from the monthly sales totals, calculates the first quarter revenue based on the first quarter sales, and finally saves the first quarter revenue results to the database.

Example Language: C *(Bad)*

```
#define JAN 1
#define FEB 2
#define MAR 3
short getMonthlySales(int month) {...}
float calculateRevenueForQuarter(short quarterSold) {...}
int determineFirstQuarterRevenue() {
   // Variable for sales revenue for the quarter
   float quarterRevenue = 0.0f;
   short JanSold = getMonthlySales(JAN); /* Get sales in January */
   short FebSold = getMonthlySales(FEB); /* Get sales in February */
   short MarSold = getMonthlySales(MAR); /* Get sales in March */
   // Calculate quarterly total
   short quarterSold = JanSold + FebSold + MarSold;
   // Calculate the total revenue for the quarter
   quarterRevenue = calculateRevenueForQuarter(quarterSold);
   saveFirstQuarterRevenue(quarterRevenue);
   return 0;
}
```

However, in this example the primitive type short int is used for both the monthly and the quarterly sales variables. In C the short int primitive type has a maximum value of 32768. This creates a potential integer overflow if the value for the three monthly sales adds up to more than the maximum value for the short int primitive type. An integer overflow can lead to data corruption, unexpected behavior, infinite loops and system crashes. To correct the situation the appropriate primitive type should be used, as in the example below, and/or provide some validation mechanism to ensure that the maximum value for the primitive type is not exceeded.

Example Language: C *(Good)*

```
...
float calculateRevenueForQuarter(long quarterSold) {...}
```

```
int determineFirstQuarterRevenue() {
  ...
  // Calculate quarterly total
  long quarterSold = JanSold + FebSold + MarSold;
  // Calculate the total revenue for the quarter
  quarterRevenue = calculateRevenueForQuarter(quarterSold);
  ...
}
```

Note that an integer overflow could also occur if the quarterSold variable has a primitive type long but the method calculateRevenueForQuarter has a parameter of type short.

Observed Examples

Reference	Description
CVE-2021-43537	Chain: in a web browser, an unsigned 64-bit integer is forcibly cast to a 32-bit integer (CWE-681) and potentially leading to an integer overflow (CWE-190). If an integer overflow occurs, this can cause heap memory corruption (CWE-122) *https://www.cve.org/CVERecord?id=CVE-2021-43537*
CVE-2022-21668	Chain: Python library does not limit the resources used to process images that specify a very large number of bands (CWE-1284), leading to excessive memory consumption (CWE-789) or an integer overflow (CWE-190). *https://www.cve.org/CVERecord?id=CVE-2022-21668*
CVE-2022-0545	Chain: 3D renderer has an integer overflow (CWE-190) leading to write-what-where condition (CWE-123) using a crafted image. *https://www.cve.org/CVERecord?id=CVE-2022-0545*
CVE-2021-30860	Chain: improper input validation (CWE-20) leads to integer overflow (CWE-190) in mobile OS, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2021-30860*
CVE-2021-30663	Chain: improper input validation (CWE-20) leads to integer overflow (CWE-190) in mobile OS, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2021-30663*
CVE-2018-10887	Chain: unexpected sign extension (CWE-194) leads to integer overflow (CWE-190), causing an out-of-bounds read (CWE-125) *https://www.cve.org/CVERecord?id=CVE-2018-10887*
CVE-2019-1010006	Chain: compiler optimization (CWE-733) removes or modifies code used to detect integer overflow (CWE-190), allowing out-of-bounds write (CWE-787). *https://www.cve.org/CVERecord?id=CVE-2019-1010006*
CVE-2010-1866	Chain: integer overflow (CWE-190) causes a negative signed value, which later bypasses a maximum-only check (CWE-839), leading to heap-based buffer overflow (CWE-122). *https://www.cve.org/CVERecord?id=CVE-2010-1866*
CVE-2010-2753	Chain: integer overflow leads to use-after-free *https://www.cve.org/CVERecord?id=CVE-2010-2753*
CVE-2005-1513	Chain: integer overflow in securely-coded mail program leads to buffer overflow. In 2005, this was regarded as unrealistic to exploit, but in 2020, it was rediscovered to be easier to exploit due to evolutions of the technology. *https://www.cve.org/CVERecord?id=CVE-2005-1513*
CVE-2002-0391	Integer overflow via a large number of arguments. *https://www.cve.org/CVERecord?id=CVE-2002-0391*
CVE-2002-0639	Integer overflow in OpenSSH as listed in the demonstrative examples. *https://www.cve.org/CVERecord?id=CVE-2002-0639*
CVE-2005-1141	Image with large width and height leads to integer overflow. *https://www.cve.org/CVERecord?id=CVE-2005-1141*
CVE-2005-0102	Length value of -1 leads to allocation of 0 bytes and resultant heap overflow. *https://www.cve.org/CVERecord?id=CVE-2005-0102*

Reference	Description
CVE-2004-2013	Length value of -1 leads to allocation of 0 bytes and resultant heap overflow. *https://www.cve.org/CVERecord?id=CVE-2004-2013*
CVE-2017-1000121	chain: unchecked message size metadata allows integer overflow (CWE-190) leading to buffer overflow (CWE-119). *https://www.cve.org/CVERecord?id=CVE-2017-1000121*
CVE-2013-1591	Chain: an integer overflow (CWE-190) in the image size calculation causes an infinite loop (CWE-835) which sequentially allocates buffers without limits (CWE-1325) until the stack is full. *https://www.cve.org/CVERecord?id=CVE-2013-1591*

Functional Areas

- Number Processing
- Memory Management
- Counters

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	802	2010 Top 25 - Risky Resource Management	800	2354
MemberOf	C	865	2011 Top 25 - Risky Resource Management	900	2371
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1137	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 03. Numeric Types and Operations (NUM)	1133	2445
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1162	SEI CERT C Coding Standard - Guidelines 08. Memory Management (MEM)	1154	2458
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	1387	2597
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses	1425	2600

Notes

Relationship

Integer overflows can be primary to buffer overflows.

Terminology

"Integer overflow" is sometimes used to cover several types of errors, including signedness errors, or buffer overflows that involve manipulation of integer data types instead of characters. Part of the confusion results from the fact that 0xffffffff is -1 in a signed context. Other confusion also arises because of the role that integer overflows have in chains.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Integer overflow (wrap or wraparound)
7 Pernicious Kingdoms			Integer Overflow
CLASP			Integer overflow
CERT C Secure Coding	INT18-C	CWE More Abstract	Evaluate integer expressions in a larger size before comparing or assigning to that size
CERT C Secure Coding	INT30-C	CWE More Abstract	Ensure that unsigned integer operations do not wrap
CERT C Secure Coding	INT32-C	Imprecise	Ensure that operations on signed integers do not result in overflow
CERT C Secure Coding	INT35-C		Evaluate integer expressions in a larger size before comparing or assigning to that size
CERT C Secure Coding	MEM07-C	CWE More Abstract	Ensure that the arguments to calloc(), when multiplied, do not wrap
CERT C Secure Coding	MEM35-C		Allocate sufficient memory for an object
WASC	3		Integer Overflows
Software Fault Patterns	SFP1		Glitch in computation
ISA/IEC 62443	Part 3-3		Req SR 3.5
ISA/IEC 62443	Part 3-3		Req SR 7.2
ISA/IEC 62443	Part 4-1		Req SR-2
ISA/IEC 62443	Part 4-1		Req SI-2
ISA/IEC 62443	Part 4-1		Req SVV-1
ISA/IEC 62443	Part 4-1		Req SVV-3
ISA/IEC 62443	Part 4-2		Req CR 3.5
ISA/IEC 62443	Part 4-2		Req CR 7.2

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
92	Forced Integer Overflow

References

[REF-145]Yves Younan. "An overview of common programming security vulnerabilities and possible solutions". Student thesis section 5.4.3. 2003 August. < http://fort-knox.org/thesis.pdf >.

[REF-146]blexim. "Basic Integer Overflows". Phrack - Issue 60, Chapter 10. < http://www.phrack.org/issues.html?issue=60&id=10#article >.

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-106]David LeBlanc and Niels Dekker. "SafeInt". < http://safeint.codeplex.com/ >.

[REF-150]Johannes Ullrich. "Top 25 Series - Rank 17 - Integer Overflow Or Wraparound". 2010 March 8. SANS Software Security Institute. < http://software-security.sans.org/blog/2010/03/18/top-25-series-rank-17-integer-overflow-or-wraparound >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-191: Integer Underflow (Wrap or Wraparound)

Weakness ID : 191
Structure : Simple
Abstraction : Base

Description

The product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result.

Extended Description

This can happen in signed and unsigned cases.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
MemberOf	C	189	Numeric Errors	2312

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Alternate Terms

Integer underflow : "Integer underflow" is sometimes used to identify signedness errors in which an originally positive number becomes negative as a result of subtraction. However, there are cases of bad subtraction in which unsigned integers are involved, so it's not always a signedness issue. "Integer underflow" is occasionally used to describe array index errors in which the index is negative.

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) DoS: Instability *This weakness will generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.*	
Integrity	Modify Memory *If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the wrap around results in other conditions such as buffer overflows, further memory corruption may occur.*	
Confidentiality Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism *This weakness can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.*	

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

Demonstrative Examples

Example 1:

The following example subtracts from a 32 bit signed integer.

Example Language: C *(Bad)*

```c
#include <stdio.h>
#include <stdbool.h>
main (void)
{
  int i;
  i = -2147483648;
  i = i - 1;
  return 0;
}
```

The example has an integer underflow. The value of i is already at the lowest negative value possible, so after subtracting 1, the new value of i is 2147483647.

Example 2:

This code performs a stack allocation based on a length calculation.

Example Language: C *(Bad)*

```c
int a = 5, b = 6;
```

```
size_t len = a - b;
char buf[len]; // Just blows up the stack
}
```

Since a and b are declared as signed ints, the "a - b" subtraction gives a negative result (-1). However, since len is declared to be unsigned, len is cast to an extremely large positive number (on 32-bit systems - 4294967295). As a result, the buffer buf[len] declaration uses an extremely large size to allocate on the stack, very likely more than the entire computer's memory space.

Miscalculations usually will not be so obvious. The calculation will either be complicated or the result of an attacker's input to attain the negative value.

Observed Examples

Reference	Description
CVE-2004-0816	Integer underflow in firewall via malformed packet. *https://www.cve.org/CVERecord?id=CVE-2004-0816*
CVE-2004-1002	Integer underflow by packet with invalid length. *https://www.cve.org/CVERecord?id=CVE-2004-1002*
CVE-2005-0199	Long input causes incorrect length calculation. *https://www.cve.org/CVERecord?id=CVE-2005-0199*
CVE-2005-1891	Malformed icon causes integer underflow in loop counter variable. *https://www.cve.org/CVERecord?id=CVE-2005-1891*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1137	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 03. Numeric Types and Operations (NUM)	1133	2445
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Integer underflow (wrap or wraparound)
Software Fault Patterns	SFP1		Glitch in computation
CERT C Secure Coding	INT30-C	Imprecise	Ensure that unsigned integer operations do not wrap
CERT C Secure Coding	INT32-C	Imprecise	Ensure that operations on signed integers do not result in overflow

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-192: Integer Coercion Error

Weakness ID : 192
Structure : Simple
Abstraction : Variant

Description

Integer coercion refers to a set of flaws pertaining to the type casting, extension, or truncation of primitive data types.

Extended Description

Several flaws fall under the category of integer coercion errors. For the most part, these errors in and of themselves result only in availability and data integrity issues. However, in some circumstances, they may result in other, more complicated security related flaws, such as buffer overflow conditions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) DoS: Crash, Exit, or Restart *Integer coercion often leads to undefined states of execution resulting in infinite loops or crashes.*	
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands *In some cases, integer coercion errors can lead to exploitable buffer overflow conditions, resulting in the execution of arbitrary code.*	
Integrity Other	Other *Integer coercion errors result in an incorrect value being stored for the variable in question.*	

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

Potential Mitigations

Phase: Requirements

A language which throws exceptions on ambiguous data casts might be chosen.

Phase: Architecture and Design

Design objects and program flow such that multiple or complex casts are unnecessary

Phase: Implementation

Ensure that any data type casting that you must used is entirely understood in order to reduce the plausibility of error in use.

Demonstrative Examples

Example 1:

The following code is intended to read an incoming packet from a socket and extract one or more headers.

Example Language: C *(Bad)*

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;
sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;
if (numHeaders > 100) {
  ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader);
ParsePacketHeaders(packet, headers);
```

The code performs a check to make sure that the packet does not contain too many headers. However, numHeaders is defined as a signed int, so it could be negative. If the incoming packet specifies a value such as -3, then the malloc calculation will generate a negative number (say, -300 if each header can be a maximum of 100 bytes). When this result is provided to malloc(), it is first converted to a size_t type. This conversion then produces a large value such as 4294966996, which may cause malloc() to fail or to allocate an extremely large amount of memory (CWE-195). With the appropriate negative numbers, an attacker could trick malloc() into using a very small positive number, which then allocates a buffer that is much smaller than expected, potentially leading to a buffer overflow.

Example 2:

The following code reads a maximum size and performs validation on that size. It then performs a strncpy, assuming it will not exceed the boundaries of the array. While the use of "short s" is forced in this particular example, short int's are frequently used within real-world code, such as code that processes structured data.

Example Language: C *(Bad)*

```
int GetUntrustedInt () {
  return(0x0000FFFF);
}
void main (int argc, char **argv) {
  char path[256];
  char *input;
  int i;
  short s;
  unsigned int sz;
  i = GetUntrustedInt();
```

```
    s = i;
    /* s is -1 so it passes the safety check - CWE-697 */
    if (s > 256) {
        DiePainfully("go away!\n");
    }
    /* s is sign-extended and saved in sz */
    sz = s;
    /* output: i=65535, s=-1, sz=4294967295 - your mileage may vary */
    printf("i=%d, s=%d, sz=%u\n", i, s, sz);
    input = GetUserInput("Enter pathname:");
    /* strncpy interprets s as unsigned int, so it's treated as MAX_INT
    (CWE-195), enabling buffer overflow (CWE-119) */
    strncpy(path, input, s);
    path[255] = '\0'; /* don't want CWE-170 */
    printf("Path is: %s\n", path);
}
```

This code first exhibits an example of CWE-839, allowing "s" to be a negative number. When the negative short "s" is converted to an unsigned integer, it becomes an extremely large positive integer. When this converted integer is used by strncpy() it will lead to a buffer overflow (CWE-119).

Observed Examples

Reference	Description
CVE-2022-2639	Chain: integer coercion error (CWE-192) prevents a return value from indicating an error, leading to out-of-bounds write (CWE-787) *https://www.cve.org/CVERecord?id=CVE-2022-2639*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Maintenance

Within C, it might be that "coercion" is semantically different than "casting", possibly depending on whether the programmer directly specifies the conversion, or if the compiler does it implicitly. This has implications for the presentation of this entry and others, such as CWE-681, and whether there is enough of a difference for these entries to be split.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Integer coercion error
CERT C Secure Coding	INT02-C		Understand integer conversion rules
CERT C Secure Coding	INT05-C		Do not use input functions to convert character data if they cannot handle all possible inputs
CERT C Secure Coding	INT31-C	Exact	Ensure that integer conversions do not result in lost or misinterpreted data

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-193: Off-by-one Error

Weakness ID : 193
Structure : Simple
Abstraction : Base

Description

A product calculates or uses an incorrect maximum or minimum value that is 1 more, or 1 less, than the correct value.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499
CanPrecede	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293
CanPrecede	B	170	Improper Null Termination	428
CanPrecede	B	617	Reachable Assertion	1378

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

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

Nature	Type	ID	Name	Page
MemberOf	C	189	Numeric Errors	2312

Applicable Platforms

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

Alternate Terms

off-by-five : An "off-by-five" error was reported for sudo in 2002 (CVE-2002-0184), but that is more like a "length calculation" error.

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) DoS: Instability	

Scope	Impact	Likelihood
	This weakness will generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.	
Integrity	Modify Memory *If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the wrap around results in other conditions such as buffer overflows, further memory corruption may occur.*	
Confidentiality Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism *This weakness can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.*	

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

Potential Mitigations

Phase: Implementation

When copying character arrays or using character manipulation methods, the correct size parameter must be used to account for the null terminator that needs to be added at the end of the array. Some examples of functions susceptible to this weakness in C include strcpy(), strncpy(), strcat(), strncat(), printf(), sprintf(), scanf() and sscanf().

Demonstrative Examples

Example 1:

The following code allocates memory for a maximum number of widgets. It then gets a user-specified number of widgets, making sure that the user does not request too many. It then initializes the elements of the array using InitializeWidget(). Because the number of widgets can vary for each request, the code inserts a NULL pointer to signify the location of the last widget.

Example Language: C *(Bad)*

```
int i;
unsigned int numWidgets;
Widget **WidgetList;
numWidgets = GetUntrustedSizeValue();
if ((numWidgets == 0) || (numWidgets > MAX_NUM_WIDGETS)) {
  ExitError("Incorrect number of widgets requested!");
}
WidgetList = (Widget **)malloc(numWidgets * sizeof(Widget *));
printf("WidgetList ptr=%p\n", WidgetList);
for(i=0; i<numWidgets; i++) {
  WidgetList[i] = InitializeWidget();
}
WidgetList[numWidgets] = NULL;
```

```
showWidgets(WidgetList);
```

However, this code contains an off-by-one calculation error (CWE-193). It allocates exactly enough space to contain the specified number of widgets, but it does not include the space for the NULL pointer. As a result, the allocated buffer is smaller than it is supposed to be (CWE-131). So if the user ever requests MAX_NUM_WIDGETS, there is an out-of-bounds write (CWE-787) when the NULL is assigned. Depending on the environment and compilation settings, this could cause memory corruption.

Example 2:

In this example, the code does not account for the terminating null character, and it writes one byte beyond the end of the buffer.

The first call to strncat() appends up to 20 characters plus a terminating null character to fullname[]. There is plenty of allocated space for this, and there is no weakness associated with this first call. However, the second call to strncat() potentially appends another 20 characters. The code does not account for the terminating null character that is automatically added by strncat(). This terminating null character would be written one byte beyond the end of the fullname[] buffer. Therefore an off-by-one error exists with the second strncat() call, as the third argument should be 19.

Example Language: C *(Bad)*

```
char firstname[20];
char lastname[20];
char fullname[40];
fullname[0] = '\0';
strncat(fullname, firstname, 20);
strncat(fullname, lastname, 20);
```

When using a function like strncat() one must leave a free byte at the end of the buffer for a terminating null character, thus avoiding the off-by-one weakness. Additionally, the last argument to strncat() is the number of characters to append, which must be less than the remaining space in the buffer. Be careful not to just use the total size of the buffer.

Example Language: C *(Good)*

```
char firstname[20];
char lastname[20];
char fullname[40];
fullname[0] = '\0';
strncat(fullname, firstname, sizeof(fullname)-strlen(fullname)-1);
strncat(fullname, lastname, sizeof(fullname)-strlen(fullname)-1);
```

Example 3:

The Off-by-one error can also be manifested when reading characters from a character array within a for loop that has an incorrect continuation condition.

Example Language: C *(Bad)*

```
#define PATH_SIZE 60
char filename[PATH_SIZE];
for(i=0; i<=PATH_SIZE; i++) {
  char c = getc();
  if (c == 'EOF') {
    filename[i] = '\0';
  }
  filename[i] = getc();
}
```

In this case, the correct continuation condition is shown below.

Example Language: C *(Good)*

```
for(i=0; i<PATH_SIZE; i++) {
...
```

Example 4:

As another example the Off-by-one error can occur when using the sprintf library function to copy a string variable to a formatted string variable and the original string variable comes from an untrusted source. As in the following example where a local function, setFilename is used to store the value of a filename to a database but first uses sprintf to format the filename. The setFilename function includes an input parameter with the name of the file that is used as the copy source in the sprintf function. The sprintf function will copy the file name to a char array of size 20 and specifies the format of the new variable as 16 characters followed by the file extension .dat.

Example Language: C *(Bad)*

```
int setFilename(char *filename) {
  char name[20];
  sprintf(name, "%16s.dat", filename);
  int success = saveFormattedFilenameToDB(name);
  return success;
}
```

However this will cause an Off-by-one error if the original filename is exactly 16 characters or larger because the format of 16 characters with the file extension is exactly 20 characters and does not take into account the required null terminator that will be placed at the end of the string.

Observed Examples

Reference	Description
CVE-2003-0252	Off-by-one error allows remote attackers to cause a denial of service and possibly execute arbitrary code via requests that do not contain newlines. *https://www.cve.org/CVERecord?id=CVE-2003-0252*
CVE-2001-1391	Off-by-one vulnerability in driver allows users to modify kernel memory. *https://www.cve.org/CVERecord?id=CVE-2001-1391*
CVE-2002-0083	Off-by-one error allows local users or remote malicious servers to gain privileges. *https://www.cve.org/CVERecord?id=CVE-2002-0083*
CVE-2002-0653	Off-by-one buffer overflow in function usd by server allows local users to execute arbitrary code as the server user via .htaccess files with long entries. *https://www.cve.org/CVERecord?id=CVE-2002-0653*
CVE-2002-0844	Off-by-one buffer overflow in version control system allows local users to execute arbitrary code. *https://www.cve.org/CVERecord?id=CVE-2002-0844*
CVE-1999-1568	Off-by-one error in FTP server allows a remote attacker to cause a denial of service (crash) via a long PORT command. *https://www.cve.org/CVERecord?id=CVE-1999-1568*
CVE-2004-0346	Off-by-one buffer overflow in FTP server allows local users to gain privileges via a 1024 byte RETR command. *https://www.cve.org/CVERecord?id=CVE-2004-0346*
CVE-2004-0005	Multiple buffer overflows in chat client allow remote attackers to cause a denial of service and possibly execute arbitrary code. *https://www.cve.org/CVERecord?id=CVE-2004-0005*
CVE-2003-0356	Multiple off-by-one vulnerabilities in product allow remote attackers to cause a denial of service and possibly execute arbitrary code. *https://www.cve.org/CVERecord?id=CVE-2003-0356*

Reference	Description
CVE-2001-1496	Off-by-one buffer overflow in server allows remote attackers to cause a denial of service and possibly execute arbitrary code. *https://www.cve.org/CVERecord?id=CVE-2001-1496*
CVE-2004-0342	This is an interesting example that might not be an off-by-one. *https://www.cve.org/CVERecord?id=CVE-2004-0342*
CVE-2001-0609	An off-by-one enables a terminating null to be overwritten, which causes 2 strings to be merged and enable a format string. *https://www.cve.org/CVERecord?id=CVE-2001-0609*
CVE-2002-1745	Off-by-one error allows source code disclosure of files with 4 letter extensions that match an accepted 3-letter extension. *https://www.cve.org/CVERecord?id=CVE-2002-1745*
CVE-2002-1816	Off-by-one buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2002-1816*
CVE-2002-1721	Off-by-one error causes an snprintf call to overwrite a critical internal variable with a null value. *https://www.cve.org/CVERecord?id=CVE-2002-1721*
CVE-2003-0466	Off-by-one error in function used in many products leads to a buffer overflow during pathname management, as demonstrated using multiple commands in an FTP server. *https://www.cve.org/CVERecord?id=CVE-2003-0466*
CVE-2003-0625	Off-by-one error allows read of sensitive memory via a malformed request. *https://www.cve.org/CVERecord?id=CVE-2003-0625*
CVE-2006-4574	Chain: security monitoring product has an off-by-one error that leads to unexpected length values, triggering an assertion. *https://www.cve.org/CVERecord?id=CVE-2006-4574*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	741	CERT C Secure Coding Standard (2008) Chapter 8 - Characters and Strings (STR)	734	2344
MemberOf	C	875	CERT C++ Secure Coding Section 07 - Characters and Strings (STR)	868	2376
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Notes

Relationship

This is not always a buffer overflow. For example, an off-by-one error could be a factor in a partial comparison, a read from the wrong memory location, an incorrect conditional, etc.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Off-by-one Error
CERT C Secure Coding	STR31-C		Guarantee that storage for strings has sufficient space for character data and the null terminator

References

[REF-155]Halvar Flake. "Third Generation Exploits". presentation at Black Hat Europe 2001. < https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fwww.blackhat.com%2Fpresentations%2Fbh-europe-01%2Fhalvar-flake%2Fbh-europe-01-halvarflake.ppt&wdOrigin=BROWSELINK >.2023-04-07.

[REF-156]Steve Christey. "Off-by-one errors: a brief explanation". Secprog and SC-L mailing list posts. 2004 May 5. < http://marc.info/?l=secprog&m=108379742110553&w=2 >.

[REF-157]klog. "The Frame Pointer Overwrite". Phrack Issue 55, Chapter 8. 1999 September 9. < https://kaizo.org/mirrors/phrack/phrack55/P55-08 >.2023-04-07.

[REF-140]Greg Hoglund and Gary McGraw. "Exploiting Software: How to Break Code". 2004 February 7. Addison-Wesley. < https://www.amazon.com/Exploiting-Software-How-Break-Code/dp/0201786958 >.2023-04-07.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-194: Unexpected Sign Extension

Weakness ID : 194
Structure : Simple
Abstraction : Variant

Description

The product performs an operation on a number that causes it to be sign extended when it is transformed into a larger data type. When the original number is negative, this can produce unexpected values that lead to resultant weaknesses.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability Other	Read Memory Modify Memory Other *When an unexpected sign extension occurs in code that operates directly on memory buffers, such as a size value or a memory index, then it could cause the program to write or read outside the boundaries of the intended buffer. If the numeric value is associated with an application-level resource, such as a quantity or price for a product in an e-commerce site, then the sign extension could produce a value that is much higher (or lower) than the application's allowable range.*	

Potential Mitigations

Phase: Implementation

Avoid using signed variables if you don't need to represent negative values. When negative values are needed, perform validation after you save those values to larger data types, or before passing them to functions that are expecting unsigned values.

Demonstrative Examples

Example 1:

The following code reads a maximum size and performs a sanity check on that size. It then performs a strncpy, assuming it will not exceed the boundaries of the array. While the use of "short s" is forced in this particular example, short int's are frequently used within real-world code, such as code that processes structured data.

Example Language: C *(Bad)*

```
int GetUntrustedInt () {
  return(0x0000FFFF);
}
void main (int argc, char **argv) {
  char path[256];
  char *input;
  int i;
  short s;
  unsigned int sz;
  i = GetUntrustedInt();
  s = i;
  /* s is -1 so it passes the safety check - CWE-697 */
  if (s > 256) {
    DiePainfully("go away!\n");
  }
  /* s is sign-extended and saved in sz */
  sz = s;
  /* output: i=65535, s=-1, sz=4294967295 - your mileage may vary */
  printf("i=%d, s=%d, sz=%u\n", i, s, sz);
  input = GetUserInput("Enter pathname:");
  /* strncpy interprets s as unsigned int, so it's treated as MAX_INT
  (CWE-195), enabling buffer overflow (CWE-119) */
  strncpy(path, input, s);
  path[255] = '\0'; /* don't want CWE-170 */
  printf("Path is: %s\n", path);
}
```

This code first exhibits an example of CWE-839, allowing "s" to be a negative number. When the negative short "s" is converted to an unsigned integer, it becomes an extremely large positive integer. When this converted integer is used by strncpy() it will lead to a buffer overflow (CWE-119).

Observed Examples

Reference	Description
CVE-2018-10887	Chain: unexpected sign extension (CWE-194) leads to integer overflow (CWE-190), causing an out-of-bounds read (CWE-125) *https://www.cve.org/CVERecord?id=CVE-2018-10887*
CVE-1999-0234	Sign extension error produces -1 value that is treated as a command separator, enabling OS command injection. *https://www.cve.org/CVERecord?id=CVE-1999-0234*
CVE-2003-0161	Product uses "char" type for input character. When char is implemented as a signed type, ASCII value 0xFF (255), a sign extension produces a -1 value that is treated as a program-specific separator value, effectively disabling a length check and leading to a buffer overflow. This is also a multiple interpretation error. *https://www.cve.org/CVERecord?id=CVE-2003-0161*
CVE-2007-4988	chain: signed short width value in image processor is sign extended during conversion to unsigned int, which leads to integer overflow and heap-based buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2007-4988*
CVE-2006-1834	chain: signedness error allows bypass of a length check; later sign extension makes exploitation easier. *https://www.cve.org/CVERecord?id=CVE-2006-1834*
CVE-2005-2753	Sign extension when manipulating Pascal-style strings leads to integer overflow and improper memory copy. *https://www.cve.org/CVERecord?id=CVE-2005-2753*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Relationship

Sign extension errors can lead to buffer overflows and other memory-based problems. They are also likely to be factors in other weaknesses that are not based on memory operations, but rely on numeric calculation.

Maintenance

This entry is closely associated with signed-to-unsigned conversion errors (CWE-195) and other numeric errors. These relationships need to be more closely examined within CWE.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Sign extension error
Software Fault Patterns	SFP1		Glitch in computation
CERT C Secure Coding	INT31-C	CWE More Specific	Ensure that integer conversions do not result in lost or misinterpreted data

References

[REF-161]John McDonald, Mark Dowd and Justin Schuh. "C Language Issues for Application Security". 2008 January 5. < http://www.informit.com/articles/article.aspx?p=686170&seqNum=6 >.

[REF-162]Robert Seacord. "Integral Security". 2006 November 3. < https://drdobbs.com/cpp/integral-security/193501774 >.2023-04-07.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-195: Signed to Unsigned Conversion Error

Weakness ID : 195
Structure : Simple
Abstraction : Variant

Description

The product uses a signed primitive and performs a cast to an unsigned primitive, which can produce an unexpected value if the value of the signed primitive can not be represented using an unsigned primitive.

Extended Description

It is dangerous to rely on implicit casts between signed and unsigned numbers because the result can take on an unexpected value and violate assumptions made by the program.

Often, functions will return negative values to indicate a failure. When the result of a function is to be used as a size parameter, using these negative return values can have unexpected results. For example, if negative size values are passed to the standard memory copy or allocation functions they will be implicitly cast to a large unsigned value. This may lead to an exploitable buffer overflow or underflow condition.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495
CanFollow	B	839	Numeric Range Comparison Without Minimum Check	1767
CanPrecede	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State *Conversion between signed and unsigned values can lead to a variety of errors, but from a security standpoint is most commonly associated with integer overflow and buffer overflow vulnerabilities.*	

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

Demonstrative Examples

Example 1:

In this example the variable amount can hold a negative value when it is returned. Because the function is declared to return an unsigned int, amount will be implicitly converted to unsigned.

Example Language: C *(Bad)*

```
unsigned int readdata () {
  int amount = 0;
  ...
  if (result == ERROR)
  amount = -1;
  ...
  return amount;
}
```

If the error condition in the code above is met, then the return value of readdata() will be 4,294,967,295 on a system that uses 32-bit integers.

Example 2:

In this example, depending on the return value of accecssmainframe(), the variable amount can hold a negative value when it is returned. Because the function is declared to return an unsigned value, amount will be implicitly cast to an unsigned number.

Example Language: C *(Bad)*

```
unsigned int readdata () {
  int amount = 0;
  ...
  amount = accessmainframe();
  ...
  return amount;
}
```

If the return value of accessmainframe() is -1, then the return value of readdata() will be 4,294,967,295 on a system that uses 32-bit integers.

Example 3:

The following code is intended to read an incoming packet from a socket and extract one or more headers.

Example Language: C *(Bad)*

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;
sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;
if (numHeaders > 100) {
  ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader);
ParsePacketHeaders(packet, headers);
```

The code performs a check to make sure that the packet does not contain too many headers. However, numHeaders is defined as a signed int, so it could be negative. If the incoming packet specifies a value such as -3, then the malloc calculation will generate a negative number (say, -300 if each header can be a maximum of 100 bytes). When this result is provided to malloc(), it is first converted to a size_t type. This conversion then produces a large value such as 4294966996, which may cause malloc() to fail or to allocate an extremely large amount of memory (CWE-195). With the appropriate negative numbers, an attacker could trick malloc() into using a very small positive number, which then allocates a buffer that is much smaller than expected, potentially leading to a buffer overflow.

Example 4:

This example processes user input comprised of a series of variable-length structures. The first 2 bytes of input dictate the size of the structure to be processed.

Example Language: C *(Bad)*

```
char* processNext(char* strm) {
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
    process(buf);
    return strm + len;
  }
  else {
    return -1;
  }
}
```

The programmer has set an upper bound on the structure size: if it is larger than 512, the input will not be processed. The problem is that len is a signed short, so the check against the maximum structure length is done with signed values, but len is converted to an unsigned integer for the call to memcpy() and the negative bit will be extended to result in a huge value for the unsigned integer. If len is negative, then it will appear that the structure has an appropriate size (the if branch will be taken), but the amount of memory copied by memcpy() will be quite large, and the attacker will be able to overflow the stack with data in strm.

Example 5:

In the following example, it is possible to request that memcpy move a much larger segment of memory than assumed:

Example Language: C *(Bad)*

```
int returnChunkSize(void *) {
  /* if chunk info is valid, return the size of usable memory,
  * else, return -1 to indicate an error
  */
  ...
}
int main() {
  ...
  memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
  ...
}
```

If returnChunkSize() happens to encounter an error it will return -1. Notice that the return value is not checked before the memcpy operation (CWE-252), so -1 can be passed as the size argument to memcpy() (CWE-805). Because memcpy() assumes that the value is unsigned, it will be interpreted as MAXINT-1 (CWE-195), and therefore will copy far more memory than is likely available to the destination buffer (CWE-787, CWE-788).

Example 6:

This example shows a typical attempt to parse a string with an error resulting from a difference in assumptions between the caller to a function and the function's action.

Example Language: C *(Bad)*

```
int proc_msg(char *s, int msg_len)
{
// Note space at the end of the string - assume all strings have preamble with space
int pre_len = sizeof("preamble: ");
char buf[pre_len - msg_len];
... Do processing here if we get this far
}
char *s = "preamble: message\n";
char *sl = strchr(s, ':'); // Number of characters up to ':' (not including space)
int jnklen = sl == NULL ? 0 : sl - s; // If undefined pointer, use zero length
int ret_val = proc_msg ("s", jnklen); // Violate assumption of preamble length, end up with negative value, blow out stack
```

The buffer length ends up being -1, resulting in a blown out stack. The space character after the colon is included in the function calculation, but not in the caller's calculation. This, unfortunately, is not usually so obvious but exists in an obtuse series of calculations.

Observed Examples

Reference	Description
CVE-2007-4268	Chain: integer signedness error (CWE-195) passes signed comparison, leading to heap overflow (CWE-122) *https://www.cve.org/CVERecord?id=CVE-2007-4268*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Signed to unsigned conversion error

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP1		Glitch in computation
CERT C Secure Coding	INT31-C	CWE More Specific	Ensure that integer conversions do not result in lost or misinterpreted data

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-196: Unsigned to Signed Conversion Error

Weakness ID : 196
Structure : Simple
Abstraction : Variant

Description

The product uses an unsigned primitive and performs a cast to a signed primitive, which can produce an unexpected value if the value of the unsigned primitive can not be represented using a signed primitive.

Extended Description

Although less frequent an issue than signed-to-unsigned conversion, unsigned-to-signed conversion can be the perfect precursor to dangerous buffer underwrite conditions that allow attackers to move down the stack where they otherwise might not have access in a normal buffer overflow condition. Buffer underwrites occur frequently when large unsigned values are cast to signed values, and then used as indexes into a buffer or for pointer arithmetic.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495
CanAlsoBe	B	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	304
CanAlsoBe	B	124	Buffer Underwrite ('Buffer Underflow')	326

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

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

Nature	Type	ID	Name	Page
ChildOf	B	681	Incorrect Conversion between Numeric Types	1495

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart *Incorrect sign conversions generally lead to undefined behavior, and therefore crashes.*	
Integrity	Modify Memory *If a poor cast lead to a buffer overflow or similar condition, data integrity may be affected.*	
Integrity Confidentiality Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism *Improper signed-to-unsigned conversions without proper checking can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.*	

Potential Mitigations

Phase: Requirements

Choose a language which is not subject to these casting flaws.

Phase: Architecture and Design

Design object accessor functions to implicitly check values for valid sizes. Ensure that all functions which will be used as a size are checked previous to use as a size. If the language permits, throw exceptions rather than using in-band errors.

Phase: Implementation

Error check the return values of all functions. Be aware of implicit casts made, and use unsigned variables for sizes if at all possible.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Unsigned to signed conversion error
Software Fault Patterns	SFP1		Glitch in computation

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
92	Forced Integer Overflow

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-197: Numeric Truncation Error

Weakness ID : 197
Structure : Simple
Abstraction : Base

Description

Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion.

Extended Description

When a primitive is cast to a smaller primitive, the high order bits of the large value are lost in the conversion, potentially resulting in an unexpected value that is not equal to the original value. This value may be required as an index into a buffer, a loop iterator, or simply necessary state data.
In any case, the value cannot be trusted and the system will be in an undefined state. While this method may be employed viably to isolate the low bits of a value, this usage is rare, and truncation usually implies that an implementation error has occurred.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	681	Incorrect Conversion between Numeric Types	1495
CanAlsoBe	Ⓥ	192	Integer Coercion Error	482
CanAlsoBe	Ⓥ	194	Unexpected Sign Extension	491
CanAlsoBe	Ⓥ	195	Signed to Unsigned Conversion Error	494
CanAlsoBe	Ⓥ	196	Unsigned to Signed Conversion Error	498

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

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	681	Incorrect Conversion between Numeric Types	1495

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

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	681	Incorrect Conversion between Numeric Types	1495

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Memory *The true value of the data is lost and corrupted data is used.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

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

Potential Mitigations

Phase: Implementation

Ensure that no casts, implicit or explicit, take place that move from a larger size primitive or a smaller size primitive.

Demonstrative Examples

Example 1:

This example, while not exploitable, shows the possible mangling of values associated with truncation errors:

Example Language: C *(Bad)*

```
int intPrimitive;
short shortPrimitive;
intPrimitive = (int)(~((int)0) ^ (1 << (sizeof(int)*8-1)));
shortPrimitive = intPrimitive;
printf("Int MAXINT: %d\nShort MAXINT: %d\n", intPrimitive, shortPrimitive);
```

The above code, when compiled and run on certain systems, returns the following output:

Example Language: *(Result)*

```
Int MAXINT: 2147483647
Short MAXINT: -1
```

This problem may be exploitable when the truncated value is used as an array index, which can happen implicitly when 64-bit values are used as indexes, as they are truncated to 32 bits.

Example 2:

In the following Java example, the method updateSalesForProduct is part of a business application class that updates the sales information for a particular product. The method receives as arguments the product ID and the integer amount sold. The product ID is used to retrieve the total product count from an inventory object which returns the count as an integer. Before calling the method of the sales object to update the sales count the integer values are converted to The primitive type short since the method requires short type for the method arguments.

Example Language: Java *(Bad)*

```
...
// update sales database for number of product sold with product ID
public void updateSalesForProduct(String productID, int amountSold) {
  // get the total number of products in inventory database
  int productCount = inventory.getProductCount(productID);
  // convert integer values to short, the method for the
  // sales object requires the parameters to be of type short
  short count = (short) productCount;
  short sold = (short) amountSold;
  // update sales database for product
  sales.updateSalesCount(productID, count, sold);
}
...
```

However, a numeric truncation error can occur if the integer values are higher than the maximum value allowed for the primitive type short. This can cause unexpected results or loss or corruption of data. In this case the sales database may be corrupted with incorrect data. Explicit casting from a from a larger size primitive type to a smaller size primitive type should be prevented.
The following example an if statement is added to validate that the integer values less than the maximum value for the primitive type short before the explicit cast and the call to the sales method.

Example Language: Java *(Good)*

```
...
// update sales database for number of product sold with product ID
public void updateSalesForProduct(String productID, int amountSold) {
  // get the total number of products in inventory database
  int productCount = inventory.getProductCount(productID);
  // make sure that integer numbers are not greater than
  // maximum value for type short before converting
  if ((productCount < Short.MAX_VALUE) && (amountSold < Short.MAX_VALUE)) {
    // convert integer values to short, the method for the
    // sales object requires the parameters to be of type short
    short count = (short) productCount;
    short sold = (short) amountSold;
    // update sales database for product
    sales.updateSalesCount(productID, count, sold);
  else {
  // throw exception or perform other processing
    ...
  }
}
...
```

Observed Examples

Reference	Description
CVE-2020-17087	Chain: integer truncation (CWE-197) causes small buffer allocation (CWE-131) leading to out-of-bounds write (CWE-787) in kernel pool, as exploited in the wild per CISA KEV. *https://www.cve.org/CVERecord?id=CVE-2020-17087*
CVE-2009-0231	Integer truncation of length value leads to heap-based buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2009-0231*
CVE-2008-3282	Size of a particular type changes for 64-bit platforms, leading to an integer truncation in document processor causes incorrect index to be generated. *https://www.cve.org/CVERecord?id=CVE-2008-3282*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	848	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 5 - Numeric Types and Operations (NUM)	844	2363
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1137	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 03. Numeric Types and Operations (NUM)	1133	2445
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1159	SEI CERT C Coding Standard - Guidelines 05. Floating Point (FLP)	1154	2457
MemberOf	C	1163	SEI CERT C Coding Standard - Guidelines 09. Input Output (FIO)	1154	2459
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Research Gap

This weakness has traditionally been under-studied and under-reported, although vulnerabilities in popular software have been published in 2008 and 2009.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Numeric truncation error
CLASP			Truncation error
CERT C Secure Coding	FIO34-C	CWE More Abstract	Distinguish between characters read from a file and EOF or WEOF
CERT C Secure Coding	FLP34-C	CWE More Abstract	Ensure that floating point conversions are within range of the new type
CERT C Secure Coding	INT02-C		Understand integer conversion rules
CERT C Secure Coding	INT05-C		Do not use input functions to convert character data if they cannot handle all possible inputs
CERT C Secure Coding	INT31-C	CWE More Abstract	Ensure that integer conversions do not result in lost or misinterpreted data
The CERT Oracle Secure Coding Standard for Java (2011)	NUM12-J		Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data
Software Fault Patterns	SFP1		Glitch in computation

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-198: Use of Incorrect Byte Ordering

Weakness ID : 198
Structure : Simple
Abstraction : Variant

Description

The product receives input from an upstream component, but it does not account for byte ordering (e.g. big-endian and little-endian) when processing the input, causing an incorrect number or value to be used.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	188	Reliance on Data/Memory Layout	470

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State	

Detection Methods

Black Box

Because byte ordering bugs are usually very noticeable even with normal inputs, this bug is more likely to occur in rarely triggered error conditions, making them difficult to detect using black box methods.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	857	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 14 - Input Output (FIO)	844	2368
MemberOf	C	993	SFP Secondary Cluster: Incorrect Input Handling	888	2417
MemberOf	C	1147	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 13. Input Output (FIO)	1133	2450
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Research Gap

Under-reported.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Numeric Byte Ordering Error
The CERT Oracle Secure Coding Standard for Java (2011)	FIO12-J		Provide methods to read and write little-endian data

CWE-200: Exposure of Sensitive Information to an Unauthorized Actor

Weakness ID : 200

Structure : Simple
Abstraction : Class

Description

The product exposes sensitive information to an actor that is not explicitly authorized to have access to that information.

Extended Description

There are many different kinds of mistakes that introduce information exposures. The severity of the error can range widely, depending on the context in which the product operates, the type of sensitive information that is revealed, and the benefits it may provide to an attacker. Some kinds of sensitive information include:

- private, personal information, such as personal messages, financial data, health records, geographic location, or contact details
- system status and environment, such as the operating system and installed packages
- business secrets and intellectual property
- network status and configuration
- the product's own code or internal state
- metadata, e.g. logging of connections or message headers
- indirect information, such as a discrepancy between two internal operations that can be observed by an outsider

Information might be sensitive to different parties, each of which may have their own expectations for whether the information should be protected. These parties include:

- the product's own users
- people or organizations whose information is created or used by the product, even if they are not direct product users
- the product's administrators, including the admins of the system(s) and/or networks on which the product operates
- the developer

Information exposures can occur in different ways:

- the code explicitly inserts sensitive information into resources or messages that are intentionally made accessible to unauthorized actors, but should not contain the information - i.e., the information should have been "scrubbed" or "sanitized"
- a different weakness or mistake indirectly inserts the sensitive information into resources, such as a web script error revealing the full system path of the program.
- the code manages resources that intentionally contain sensitive information, but the resources are unintentionally made accessible to unauthorized actors. In this case, the information exposure is resultant - i.e., a different weakness enabled the access to the information in the first place.

It is common practice to describe any loss of confidentiality as an "information exposure," but this can lead to overuse of CWE-200 in CWE mapping. From the CWE perspective, loss of confidentiality is a technical impact that can arise from dozens of different weaknesses, such as insecure file permissions or out-of-bounds read. CWE-200 and its lower-level descendants are intended to cover the mistakes that occur in behaviors that explicitly manage, store, transfer, or cleanse sensitive information.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	668	Exposure of Resource to Wrong Sphere	1469
ParentOf	Ⓑ	201	Insertion of Sensitive Information Into Sent Data	514
ParentOf	Ⓑ	203	Observable Discrepancy	518
ParentOf	Ⓑ	209	Generation of Error Message Containing Sensitive Information	533
ParentOf	Ⓑ	213	Exposure of Sensitive Information Due to Incompatible Policies	547
ParentOf	Ⓑ	215	Insertion of Sensitive Information Into Debugging Code	551
ParentOf	Ⓑ	359	Exposure of Private Personal Information to an Unauthorized Actor	882
ParentOf	Ⓑ	497	Exposure of Sensitive System Information to an Unauthorized Control Sphere	1193
ParentOf	Ⓑ	538	Insertion of Sensitive Information into Externally-Accessible File or Directory	1248
ParentOf	Ⓑ	1258	Exposure of Sensitive System Information Due to Uncleared Debug Information	2071
ParentOf	Ⓑ	1273	Device Unlock Credential Sharing	2106
ParentOf	Ⓑ	1295	Debug Messages Revealing Unnecessary Information	2152
CanFollow	Ⓥ	498	Cloneable Class Containing Sensitive Information	1196
CanFollow	Ⓥ	499	Serializable Class Containing Sensitive Data	1198
CanFollow	Ⓑ	1272	Sensitive Information Uncleared Before Debug/Power State Transition	2104

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ParentOf	Ⓑ	203	Observable Discrepancy	518
ParentOf	Ⓑ	209	Generation of Error Message Containing Sensitive Information	533
ParentOf	Ⓑ	532	Insertion of Sensitive Information into Log File	1241

Weakness Ordinalities

Primary : Developers may insert sensitive information that they do not believe, or they might forget to remove the sensitive information after it has been processed

Resultant : Separate mistakes or weaknesses could inadvertently make the sensitive information available to an attacker, such as in a detailed error message that can be read by an unauthorized party

Applicable Platforms

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

Technology : Mobile *(Prevalence = Undetermined)*

Alternate Terms

Information Disclosure : This term is frequently used in vulnerability advisories to describe a consequence or technical impact, for any vulnerability that has a loss of confidentiality. Often, CWE-200 can be misused to represent the loss of confidentiality, even when the mistake - i.e., the weakness - is not directly related to the mishandling of the information itself, such as an out-of-bounds read that accesses sensitive memory contents; here, the out-of-bounds read is the primary weakness, not the disclosure of the memory. In addition, this phrase is also used frequently in policies and legal documents, but it does not refer to any disclosure of security-relevant information.

Information Leak : This is a frequently used term, however the "leak" term has multiple uses within security. In some cases it deals with the accidental exposure of information from a different weakness, but in other cases (such as "memory leak"), this deals with improper tracking of resources, which can lead to exhaustion. As a result, CWE is actively avoiding usage of the "leak" term.

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

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 Inter-application Flow Analysis

Effectiveness = SOAR Partial

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful: Highly cost effective: Web Application Scanner Web Services Scanner Database Scanners

Effectiveness = High

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Fuzz Tester Framework-based Fuzzer Automated Monitored Execution Monitored Virtual Environment - run potentially malicious code in sandbox / wrapper / virtual machine, see if it does anything suspicious

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: 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: Context-configured Source Code Weakness Analyzer Cost effective for partial coverage: Source code Weakness Analyzer

Effectiveness = High

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Attack Modeling Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Strategy = Separation of Privilege

Compartmentalize the system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area. Ensure that appropriate compartmentalization is built into the system design, and the compartmentalization allows for and reinforces privilege separation functionality. Architects and designers should rely on the principle of least privilege to decide the appropriate time to use privileges and the time to drop privileges.

Demonstrative Examples

Example 1:

The following code checks validity of the supplied username and password and notifies the user of a successful or failed login.

Example Language: Perl *(Bad)*

```
my $username=param('username');
my $password=param('password');
if (IsValidUsername($username) == 1)
{
   if (IsValidPassword($username, $password) == 1)
   {
      print "Login Successful";
   }
   else
   {
      print "Login Failed - incorrect password";
   }
}
else
{
   print "Login Failed - unknown username";
}
```

In the above code, there are different messages for when an incorrect username is supplied, versus when the username is correct but the password is wrong. This difference enables a potential attacker to understand the state of the login function, and could allow an attacker to discover a valid username by trying different values until the incorrect password message is returned. In essence, this makes it easier for an attacker to obtain half of the necessary authentication credentials.

While this type of information may be helpful to a user, it is also useful to a potential attacker. In the above example, the message for both failed cases should be the same, such as:

Example Language: *(Result)*

```
"Login Failed - incorrect username or password"
```

Example 2:

This code tries to open a database connection, and prints any exceptions that occur.

Example Language: PHP *(Bad)*

```
try {
   openDbConnection();
}
//print exception message that includes exception message and configuration file location
catch (Exception $e) {
   echo 'Caught exception: ', $e->getMessage(), '\n';
   echo 'Check credentials in config file at: ', $Mysql_config_location, '\n';
}
```

If an exception occurs, the printed message exposes the location of the configuration file the script is using. An attacker can use this information to target the configuration file (perhaps exploiting a Path Traversal weakness). If the file can be read, the attacker could gain credentials for accessing the database. The attacker may also be able to replace the file with a malicious one, causing the application to use an arbitrary database.

Example 3:

In the example below, the method getUserBankAccount retrieves a bank account object from a database using the supplied username and account number to query the database. If an SQLException is raised when querying the database, an error message is created and output to a log file.

Example Language: Java *(Bad)*

```
public BankAccount getUserBankAccount(String username, String accountNumber) {
  BankAccount userAccount = null;
  String query = null;
  try {
    if (isAuthorizedUser(username)) {
      query = "SELECT * FROM accounts WHERE owner = "
      + username + " AND accountID = " + accountNumber;
      DatabaseManager dbManager = new DatabaseManager();
      Connection conn = dbManager.getConnection();
      Statement stmt = conn.createStatement();
      ResultSet queryResult = stmt.executeQuery(query);
      userAccount = (BankAccount)queryResult.getObject(accountNumber);
    }
  } catch (SQLException ex) {
    String logMessage = "Unable to retrieve account information from database,\nquery: " + query;
    Logger.getLogger(BankManager.class.getName()).log(Level.SEVERE, logMessage, ex);
  }
  return userAccount;
}
```

The error message that is created includes information about the database query that may contain sensitive information about the database or query logic. In this case, the error message will expose the table name and column names used in the database. This data could be used to simplify other attacks, such as SQL injection (CWE-89) to directly access the database.

Example 4:

This code stores location information about the current user:

Example Language: Java *(Bad)*

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());
...
catch (Exception e) {
  AlertDialog.Builder builder = new AlertDialog.Builder(this);
  builder.setMessage("Sorry, this application has experienced an error.");
  AlertDialog alert = builder.create();
  alert.show();
  Log.e("ExampleActivity", "Caught exception: " + e + " While on User:" + User.toString());
}
```

When the application encounters an exception it will write the user object to the log. Because the user object contains location information, the user's location is also written to the log.

Example 5:

The following is an actual MySQL error statement:

Example Language: SQL *(Result)*

```
Warning: mysql_pconnect(): Access denied for user: 'root@localhost' (Using password: N1nj4) in /usr/local/www/wi-data/
includes/database.inc on line 4
```

The error clearly exposes the database credentials.

Example 6:

This code displays some information on a web page.

Example Language: JSP *(Bad)*

```
Social Security Number: <%= ssn %></br>Credit Card Number: <%= ccn %>
```

The code displays a user's credit card and social security numbers, even though they aren't absolutely necessary.

Example 7:

The following program changes its behavior based on a debug flag.

Example Language: JSP *(Bad)*

```
<% if (Boolean.getBoolean("debugEnabled")) {
  %>
  User account number: <%= acctNo %>
  <%
  } %>
```

The code writes sensitive debug information to the client browser if the "debugEnabled" flag is set to true .

Example 8:

This code uses location to determine the user's current US State location.

First the application must declare that it requires the ACCESS_FINE_LOCATION permission in the application's manifest.xml:

Example Language: XML *(Bad)*

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

During execution, a call to getLastLocation() will return a location based on the application's location permissions. In this case the application has permission for the most accurate location possible:

Example Language: Java *(Bad)*

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
Location userCurrLocation;
userCurrLocation = locationClient.getLastLocation();
deriveStateFromCoords(userCurrLocation);
```

While the application needs this information, it does not need to use the ACCESS_FINE_LOCATION permission, as the ACCESS_COARSE_LOCATION permission will be sufficient to identify which US state the user is in.

Observed Examples

Reference	Description
CVE-2022-31162	Rust library leaks Oauth client details in application debug logs *https://www.cve.org/CVERecord?id=CVE-2022-31162*
CVE-2021-25476	Digital Rights Management (DRM) capability for mobile platform leaks pointer information, simplifying ASLR bypass *https://www.cve.org/CVERecord?id=CVE-2021-25476*
CVE-2001-1483	Enumeration of valid usernames based on inconsistent responses *https://www.cve.org/CVERecord?id=CVE-2001-1483*
CVE-2001-1528	Account number enumeration via inconsistent responses. *https://www.cve.org/CVERecord?id=CVE-2001-1528*
CVE-2004-2150	User enumeration via discrepancies in error messages. *https://www.cve.org/CVERecord?id=CVE-2004-2150*
CVE-2005-1205	Telnet protocol allows servers to obtain sensitive environment information from clients. *https://www.cve.org/CVERecord?id=CVE-2005-1205*
CVE-2002-1725	Script calls phpinfo(), revealing system configuration to web user *https://www.cve.org/CVERecord?id=CVE-2002-1725*
CVE-2002-0515	Product sets a different TTL when a port is being filtered than when it is not being filtered, which allows remote attackers to identify filtered ports by comparing TTLs. *https://www.cve.org/CVERecord?id=CVE-2002-0515*
CVE-2004-0778	Version control system allows remote attackers to determine the existence of arbitrary files and directories via the -X command for an alternate history file, which causes different error messages to be returned. *https://www.cve.org/CVERecord?id=CVE-2004-0778*
CVE-2000-1117	Virtual machine allows malicious web site operators to determine the existence of files on the client by measuring delays in the execution of the getSystemResource method. *https://www.cve.org/CVERecord?id=CVE-2000-1117*
CVE-2003-0190	Product immediately sends an error message when a user does not exist, which allows remote attackers to determine valid usernames via a timing attack. *https://www.cve.org/CVERecord?id=CVE-2003-0190*
CVE-2008-2049	POP3 server reveals a password in an error message after multiple APOP commands are sent. Might be resultant from another weakness. *https://www.cve.org/CVERecord?id=CVE-2008-2049*
CVE-2007-5172	Program reveals password in error message if attacker can trigger certain database errors. *https://www.cve.org/CVERecord?id=CVE-2007-5172*
CVE-2008-4638	Composite: application running with high privileges (CWE-250) allows user to specify a restricted file to process, which generates a parsing error that leaks the contents of the file (CWE-209). *https://www.cve.org/CVERecord?id=CVE-2008-4638*
CVE-2007-1409	Direct request to library file in web application triggers pathname leak in error message. *https://www.cve.org/CVERecord?id=CVE-2007-1409*
CVE-2005-0603	Malformed regexp syntax leads to information exposure in error message. *https://www.cve.org/CVERecord?id=CVE-2005-0603*
CVE-2004-2268	Password exposed in debug information. *https://www.cve.org/CVERecord?id=CVE-2004-2268*
CVE-2003-1078	FTP client with debug option enabled shows password to the screen. *https://www.cve.org/CVERecord?id=CVE-2003-1078*
CVE-2022-0708	Collaboration platform does not clear team emails in a response, allowing leak of email addresses

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2022-0708

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	635	Weaknesses Originally Used by NVD from 2008 to 2016	635	2552
MemberOf	C	717	OWASP Top Ten 2007 Category A6 - Information Leakage and Improper Error Handling	629	2332
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	V	1003	Weaknesses for Simplified Mapping of Published Vulnerabilities	1003	2576
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Notes

Maintenance

As a result of mapping analysis in the 2020 Top 25 and more recent versions, this weakness is under review, since it is frequently misused in mapping to cover many problems that lead to loss of confidentiality. See Mapping Notes, Extended Description, and Alternate Terms.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Information Leak (information disclosure)
OWASP Top Ten 2007	A6	CWE More Specific	Information Leakage and Improper Error Handling
WASC	13		Information Leakage

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
13	Subverting Environment Variable Values
22	Exploiting Trust in Client
59	Session Credential Falsification through Prediction
60	Reusing Session IDs (aka Session Replay)
79	Using Slashes in Alternate Encoding
116	Excavation
169	Footprinting
224	Fingerprinting
285	ICMP Echo Request Ping
287	TCP SYN Scan
290	Enumerate Mail Exchange (MX) Records
291	DNS Zone Transfers

CAPEC-ID	Attack Pattern Name
292	Host Discovery
293	Traceroute Route Enumeration
294	ICMP Address Mask Request
295	Timestamp Request
296	ICMP Information Request
297	TCP ACK Ping
298	UDP Ping
299	TCP SYN Ping
300	Port Scanning
301	TCP Connect Scan
302	TCP FIN Scan
303	TCP Xmas Scan
304	TCP Null Scan
305	TCP ACK Scan
306	TCP Window Scan
307	TCP RPC Scan
308	UDP Scan
309	Network Topology Mapping
310	Scanning for Vulnerable Software
312	Active OS Fingerprinting
313	Passive OS Fingerprinting
317	IP ID Sequencing Probe
318	IP 'ID' Echoed Byte-Order Probe
319	IP (DF) 'Don't Fragment Bit' Echoing Probe
320	TCP Timestamp Probe
321	TCP Sequence Number Probe
322	TCP (ISN) Greatest Common Divisor Probe
323	TCP (ISN) Counter Rate Probe
324	TCP (ISN) Sequence Predictability Probe
325	TCP Congestion Control Flag (ECN) Probe
326	TCP Initial Window Size Probe
327	TCP Options Probe
328	TCP 'RST' Flag Checksum Probe
329	ICMP Error Message Quoting Probe
330	ICMP Error Message Echoing Integrity Probe
472	Browser Fingerprinting
497	File Discovery
508	Shoulder Surfing
573	Process Footprinting
574	Services Footprinting
575	Account Footprinting
576	Group Permission Footprinting
577	Owner Footprinting
616	Establish Rogue Location
643	Identify Shared Files/Directories on System
646	Peripheral Footprinting
651	Eavesdropping

References

[REF-172]Chris Wysopal. "Mobile App Top 10 List". 2010 December 3. < https://www.veracode.com/blog/2010/12/mobile-app-top-10-list >.2023-04-07.

[REF-1287]MITRE. "Supplemental Details - 2022 CWE Top 25". 2022 June 8. < https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25_supplemental.html#problematicMappingDetails >.

CWE-201: Insertion of Sensitive Information Into Sent Data

Weakness ID : 201
Structure : Simple
Abstraction : Base

Description

The code transmits data to another actor, but a portion of the data includes sensitive information that should not be accessible to that actor.

Extended Description

Sensitive information could include data that is sensitive in and of itself (such as credentials or private messages), or otherwise useful in the further exploitation of the system (such as internal file system structure).

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	C	200	Exposure of Sensitive Information to an Unauthorized Actor	504
ParentOf	V	598	Use of GET Request Method With Sensitive Query Strings	1340
CanAlsoBe	B	202	Exposure of Sensitive Information Through Data Queries	516
CanAlsoBe	B	209	Generation of Error Message Containing Sensitive Information	533
CanFollow	B	212	Improper Removal of Sensitive Information Before Storage or Transfer	544
CanFollow	B	226	Sensitive Information in Resource Not Removed Before Reuse	562

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1015	Limit Access	2430

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

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Files or Directories Read Memory Read Application Data *Sensitive data may be exposed to attackers.*	

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

Potential Mitigations

Phase: Requirements

Specify which data in the software should be regarded as sensitive. Consider which types of users should have access to which types of data.

Phase: Implementation

Ensure that any possibly sensitive data specified in the requirements is verified with designers to ensure that it is either a calculated risk or mitigated elsewhere. Any information that is not necessary to the functionality should be removed in order to lower both the overhead and the possibility of security sensitive data being sent.

Phase: System Configuration

Setup default error messages so that unexpected errors do not disclose sensitive information.

Phase: Architecture and Design

Strategy = Separation of Privilege

Compartmentalize the system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area. Ensure that appropriate compartmentalization is built into the system design, and the compartmentalization allows for and reinforces privilege separation functionality. Architects and designers should rely on the principle of least privilege to decide the appropriate time to use privileges and the time to drop privileges.

Demonstrative Examples

Example 1:

The following is an actual MySQL error statement:

Example Language: SQL *(Result)*

```
Warning: mysql_pconnect(): Access denied for user: 'root@localhost' (Using password: N1nj4) in /usr/local/www/wi-data/
includes/database.inc on line 4
```

The error clearly exposes the database credentials.

Observed Examples

Reference	Description
CVE-2022-0708	Collaboration platform does not clear team emails in a response, allowing leak of email addresses *https://www.cve.org/CVERecord?id=CVE-2022-0708*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Accidental leaking of sensitive information through sent data

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
12	Choosing Message Identifier
217	Exploiting Incorrectly Configured SSL/TLS
612	WiFi MAC Address Tracking
613	WiFi SSID Tracking
618	Cellular Broadcast Message Request
619	Signal Strength Tracking
621	Analysis of Packet Timing and Sizes
622	Electromagnetic Side-Channel Attack
623	Compromising Emanations Attack

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-202: Exposure of Sensitive Information Through Data Queries

Weakness ID : 202
Structure : Simple
Abstraction : Base

Description

When trying to keep information confidential, an attacker can often infer some of the information by using statistics.

Extended Description

In situations where data should not be tied to individual users, but a large number of users should be able to make queries that "scrub" the identity of users, it may be possible to get information about a user -- e.g., by specifying search terms that are known to be unique to that user.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	1230	Exposure of Sensitive Information Through Metadata	2006

Applicable Platforms

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

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Files or Directories Read Application Data *Sensitive information may possibly be leaked through data queries accidentally.*	

Potential Mitigations

Phase: Architecture and Design

This is a complex topic. See the book Translucent Databases for a good discussion of best practices.

Demonstrative Examples

Example 1:

See the book Translucent Databases for examples.

Observed Examples

Reference	Description
CVE-2022-41935	Wiki product allows an adversary to discover filenames via a series of queries starting with one letter and then iteratively extending the match. *https://www.cve.org/CVERecord?id=CVE-2022-41935*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Notes

Maintenance

The relationship between CWE-202 and CWE-612 needs to be investigated more closely, as they may be different descriptions of the same kind of problem. CWE-202 is also being considered for deprecation, as it is not clearly described and may have been misunderstood by CWE users. It could be argued that this issue is better covered by CAPEC; an attacker can utilize their data-query privileges to perform this kind of operation, and if the attacker should not be allowed to perform the operation - or if the sensitive data should not have been made accessible at all - then that is more appropriately classified as a separate CWE related to authorization (see the parent, CWE-1230).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Accidental leaking of sensitive information through data queries

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-203: Observable Discrepancy

Weakness ID : 203
Structure : Simple
Abstraction : Base

Description

The product behaves differently or sends different responses under different circumstances in a way that is observable to an unauthorized actor, which exposes security-relevant information about the state of the product, such as whether a particular operation was successful or not.

Extended Description

Discrepancies can take many forms, and variations may be detectable in timing, control flow, communications such as replies or requests, or general behavior. These discrepancies can reveal information about the product's operation or internal state to an unauthorized actor. In some cases, discrepancies can be used by attackers to form a side channel.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	200	Exposure of Sensitive Information to an Unauthorized Actor	504
ParentOf	Ⓑ	204	Observable Response Discrepancy	523
ParentOf	Ⓑ	205	Observable Behavioral Discrepancy	526
ParentOf	Ⓑ	208	Observable Timing Discrepancy	529
ParentOf	Ⓑ	1300	Improper Protection of Physical Side Channels	2165
ParentOf	Ⓑ	1303	Non-Transparent Sharing of Microarchitectural Resources	2174

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	200	Exposure of Sensitive Information to an Unauthorized Actor	504

Relevant to the view "Hardware Design" (CWE-1194)

Nature	Type	ID	Name	Page
ParentOf	Ⓑ	1300	Improper Protection of Physical Side Channels	2165

Applicable Platforms

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

Technology : Not Technology-Specific *(Prevalence = Undetermined)*

Alternate Terms

Side Channel Attack : Observable Discrepancies are at the root of side channel attacks.

Common Consequences

Scope	Impact	Likelihood
Confidentiality Access Control	Read Application Data Bypass Protection Mechanism *An attacker can gain access to sensitive information about the system, including authentication information that may allow an attacker to gain access to the system.*	

Scope	Impact	Likelihood
Confidentiality	Read Application Data *When cryptographic primitives are vulnerable to side-channel-attacks, this could be used to reveal unencrypted plaintext in the worst case.*	

Potential Mitigations

Phase: Architecture and Design

Strategy = Separation of Privilege

Compartmentalize the system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area. Ensure that appropriate compartmentalization is built into the system design, and the compartmentalization allows for and reinforces privilege separation functionality. Architects and designers should rely on the principle of least privilege to decide the appropriate time to use privileges and the time to drop privileges.

Phase: 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.

Demonstrative Examples

Example 1:

The following code checks validity of the supplied username and password and notifies the user of a successful or failed login.

Example Language: Perl *(Bad)*

```
my $username=param('username');
my $password=param('password');
if (IsValidUsername($username) == 1)
{
   if (IsValidPassword($username, $password) == 1)
   {
      print "Login Successful";
   }
   else
   {
      print "Login Failed - incorrect password";
   }
}
else
{
   print "Login Failed - unknown username";
}
```

In the above code, there are different messages for when an incorrect username is supplied, versus when the username is correct but the password is wrong. This difference enables a potential attacker to understand the state of the login function, and could allow an attacker to discover a valid username by trying different values until the incorrect password message is returned. In essence, this makes it easier for an attacker to obtain half of the necessary authentication credentials.

While this type of information may be helpful to a user, it is also useful to a potential attacker. In the above example, the message for both failed cases should be the same, such as:

Example Language: *(Result)*

```
"Login Failed - incorrect username or password"
```

Example 2:

In this example, the attacker observes how long an authentication takes when the user types in the correct password.

When the attacker tries their own values, they can first try strings of various length. When they find a string of the right length, the computation will take a bit longer, because the for loop will run at least once. Additionally, with this code, the attacker can possibly learn one character of the password at a time, because when they guess the first character right, the computation will take longer than a wrong guesses. Such an attack can break even the most sophisticated password with a few hundred guesses.

Example Language: Python *(Bad)*

```python
def validate_password(actual_pw, typed_pw):
  if len(actual_pw) <> len(typed_pw):
    return 0
  for i in len(actual_pw):
    if actual_pw[i] <> typed_pw[i]:
      return 0
  return 1
```

Note that in this example, the actual password must be handled in constant time as far as the attacker is concerned, even if the actual password is of an unusual length. This is one reason why it is good to use an algorithm that, among other things, stores a seeded cryptographic one-way hash of the password, then compare the hashes, which will always be of the same length.

Example 3:

Non-uniform processing time causes timing channel.

Example Language: *(Bad)*

Suppose an algorithm for implementing an encryption routine works fine per se, but the time taken to output the result of the encryption routine depends on a relationship between the input plaintext and the key (e.g., suppose, if the plaintext is similar to the key, it would run very fast).

In the example above, an attacker may vary the inputs, then observe differences between processing times (since different plaintexts take different time). This could be used to infer information about the key.

Example Language: *(Good)*

Artificial delays may be added to ensured all calculations take equal time to execute.

Example 4:

Suppose memory access patterns for an encryption routine are dependent on the secret key.

An attacker can recover the key by knowing if specific memory locations have been accessed or not. The value stored at those memory locations is irrelevant. The encryption routine's memory accesses will affect the state of the processor cache. If cache resources are shared across contexts, after the encryption routine completes, an attacker in different execution context can discover which memory locations the routine accessed by measuring the time it takes for their own memory accesses to complete.

Observed Examples

Reference	Description
CVE-2020-8695	Observable discrepancy in the RAPL interface for some Intel processors allows information disclosure. *https://www.cve.org/CVERecord?id=CVE-2020-8695*
CVE-2019-14353	Crypto hardware wallet's power consumption relates to total number of pixels illuminated, creating a side channel in the USB connection that allows attackers to determine secrets displayed such as PIN numbers and passwords *https://www.cve.org/CVERecord?id=CVE-2019-14353*
CVE-2019-10071	Java-oriented framework compares HMAC signatures using String.equals() instead of a constant-time algorithm, causing timing discrepancies *https://www.cve.org/CVERecord?id=CVE-2019-10071*
CVE-2002-2094	This, and others, use ".." attacks and monitor error responses, so there is overlap with directory traversal. *https://www.cve.org/CVERecord?id=CVE-2002-2094*
CVE-2001-1483	Enumeration of valid usernames based on inconsistent responses *https://www.cve.org/CVERecord?id=CVE-2001-1483*
CVE-2001-1528	Account number enumeration via inconsistent responses. *https://www.cve.org/CVERecord?id=CVE-2001-1528*
CVE-2004-2150	User enumeration via discrepancies in error messages. *https://www.cve.org/CVERecord?id=CVE-2004-2150*
CVE-2005-1650	User enumeration via discrepancies in error messages. *https://www.cve.org/CVERecord?id=CVE-2005-1650*
CVE-2004-0294	Bulletin Board displays different error messages when a user exists or not, which makes it easier for remote attackers to identify valid users and conduct a brute force password guessing attack. *https://www.cve.org/CVERecord?id=CVE-2004-0294*
CVE-2004-0243	Operating System, when direct remote login is disabled, displays a different message if the password is correct, which allows remote attackers to guess the password via brute force methods. *https://www.cve.org/CVERecord?id=CVE-2004-0243*
CVE-2002-0514	Product allows remote attackers to determine if a port is being filtered because the response packet TTL is different than the default TTL. *https://www.cve.org/CVERecord?id=CVE-2002-0514*
CVE-2002-0515	Product sets a different TTL when a port is being filtered than when it is not being filtered, which allows remote attackers to identify filtered ports by comparing TTLs. *https://www.cve.org/CVERecord?id=CVE-2002-0515*
CVE-2002-0208	Product modifies TCP/IP stack and ICMP error messages in unusual ways that show the product is in use. *https://www.cve.org/CVERecord?id=CVE-2002-0208*
CVE-2004-2252	Behavioral infoleak by responding to SYN-FIN packets. *https://www.cve.org/CVERecord?id=CVE-2004-2252*
CVE-2001-1387	Product may generate different responses than specified by the administrator, possibly leading to an information leak. *https://www.cve.org/CVERecord?id=CVE-2001-1387*
CVE-2004-0778	Version control system allows remote attackers to determine the existence of arbitrary files and directories via the -X command for an alternate history file, which causes different error messages to be returned. *https://www.cve.org/CVERecord?id=CVE-2004-0778*

Reference	Description
CVE-2004-1428	FTP server generates an error message if the user name does not exist instead of prompting for a password, which allows remote attackers to determine valid usernames. *https://www.cve.org/CVERecord?id=CVE-2004-1428*
CVE-2003-0078	SSL implementation does not perform a MAC computation if an incorrect block cipher padding is used, which causes an information leak (timing discrepancy) that may make it easier to launch cryptographic attacks that rely on distinguishing between padding and MAC verification errors, possibly leading to extraction of the original plaintext, aka the "Vaudenay timing attack." *https://www.cve.org/CVERecord?id=CVE-2003-0078*
CVE-2000-1117	Virtual machine allows malicious web site operators to determine the existence of files on the client by measuring delays in the execution of the getSystemResource method. *https://www.cve.org/CVERecord?id=CVE-2000-1117*
CVE-2003-0637	Product uses a shorter timeout for a non-existent user than a valid user, which makes it easier for remote attackers to guess usernames and conduct brute force password guessing. *https://www.cve.org/CVERecord?id=CVE-2003-0637*
CVE-2003-0190	Product immediately sends an error message when a user does not exist, which allows remote attackers to determine valid usernames via a timing attack. *https://www.cve.org/CVERecord?id=CVE-2003-0190*
CVE-2004-1602	FTP server responds in a different amount of time when a given username exists, which allows remote attackers to identify valid usernames by timing the server response. *https://www.cve.org/CVERecord?id=CVE-2004-1602*
CVE-2005-0918	Browser allows remote attackers to determine the existence of arbitrary files by setting the src property to the target filename and using Javascript to determine if the web page immediately stops loading, which indicates whether the file exists or not. *https://www.cve.org/CVERecord?id=CVE-2005-0918*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	717	OWASP Top Ten 2007 Category A6 - Information Leakage and Improper Error Handling	629	2332
MemberOf	C	728	OWASP Top Ten 2004 Category A7 - Improper Error Handling	711	2337
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1205	Security Primitives and Cryptography Issues	1194	2473
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Discrepancy Information Leaks
OWASP Top Ten 2007	A6	CWE More Specific	Information Leakage and Improper Error Handling
OWASP Top Ten 2004	A7	CWE More Specific	Improper Error Handling

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
189	Black Box Reverse Engineering

CWE-204: Observable Response Discrepancy

Weakness ID : 204
Structure : Simple
Abstraction : Base

Description

The product provides different responses to incoming requests in a way that reveals internal state information to an unauthorized actor outside of the intended control sphere.

Extended Description

This issue frequently occurs during authentication, where a difference in failed-login messages could allow an attacker to determine if the username is valid or not. These exposures can be inadvertent (bug) or intentional (design).

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	203	Observable Discrepancy	518

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

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality Access Control	Read Application Data Bypass Protection Mechanism	

Potential Mitigations

Phase: Architecture and Design

Strategy = Separation of Privilege

Compartmentalize the system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area. Ensure that appropriate compartmentalization is built into the system design, and the compartmentalization allows for and reinforces privilege separation functionality. Architects and designers should rely on the principle of least privilege to decide the appropriate time to use privileges and the time to drop privileges.

Phase: 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.

Demonstrative Examples

Example 1:

The following code checks validity of the supplied username and password and notifies the user of a successful or failed login.

Example Language: Perl *(Bad)*

```
my $username=param('username');
my $password=param('password');
if (IsValidUsername($username) == 1)
{
  if (IsValidPassword($username, $password) == 1)
  {
    print "Login Successful";
  }
  else
  {
    print "Login Failed - incorrect password";
  }
}
else
{
  print "Login Failed - unknown username";
}
```

In the above code, there are different messages for when an incorrect username is supplied, versus when the username is correct but the password is wrong. This difference enables a potential attacker to understand the state of the login function, and could allow an attacker to discover a valid username by trying different values until the incorrect password message is returned. In essence, this makes it easier for an attacker to obtain half of the necessary authentication credentials.

While this type of information may be helpful to a user, it is also useful to a potential attacker. In the above example, the message for both failed cases should be the same, such as:

Example Language: *(Result)*

```
"Login Failed - incorrect username or password"
```

Observed Examples

Reference	Description
CVE-2002-2094	This, and others, use ".." attacks and monitor error responses, so there is overlap with directory traversal. *https://www.cve.org/CVERecord?id=CVE-2002-2094*
CVE-2001-1483	Enumeration of valid usernames based on inconsistent responses *https://www.cve.org/CVERecord?id=CVE-2001-1483*
CVE-2001-1528	Account number enumeration via inconsistent responses. *https://www.cve.org/CVERecord?id=CVE-2001-1528*
CVE-2004-2150	User enumeration via discrepancies in error messages. *https://www.cve.org/CVERecord?id=CVE-2004-2150*
CVE-2005-1650	User enumeration via discrepancies in error messages.

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2005-1650
CVE-2004-0294	Bulletin Board displays different error messages when a user exists or not, which makes it easier for remote attackers to identify valid users and conduct a brute force password guessing attack. *https://www.cve.org/CVERecord?id=CVE-2004-0294*
CVE-2004-0243	Operating System, when direct remote login is disabled, displays a different message if the password is correct, which allows remote attackers to guess the password via brute force methods. *https://www.cve.org/CVERecord?id=CVE-2004-0243*
CVE-2002-0514	Product allows remote attackers to determine if a port is being filtered because the response packet TTL is different than the default TTL. *https://www.cve.org/CVERecord?id=CVE-2002-0514*
CVE-2002-0515	Product sets a different TTL when a port is being filtered than when it is not being filtered, which allows remote attackers to identify filtered ports by comparing TTLs. *https://www.cve.org/CVERecord?id=CVE-2002-0515*
CVE-2001-1387	Product may generate different responses than specified by the administrator, possibly leading to an information leak. *https://www.cve.org/CVERecord?id=CVE-2001-1387*
CVE-2004-0778	Version control system allows remote attackers to determine the existence of arbitrary files and directories via the -X command for an alternate history file, which causes different error messages to be returned. *https://www.cve.org/CVERecord?id=CVE-2004-0778*
CVE-2004-1428	FTP server generates an error message if the user name does not exist instead of prompting for a password, which allows remote attackers to determine valid usernames. *https://www.cve.org/CVERecord?id=CVE-2004-1428*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Notes

Relationship

can overlap errors related to escalated privileges

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Response discrepancy infoleak

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
331	ICMP IP Total Length Field Probe
332	ICMP IP 'ID' Field Error Message Probe
541	Application Fingerprinting
580	System Footprinting

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-205: Observable Behavioral Discrepancy

Weakness ID : 205
Structure : Simple
Abstraction : Base

Description

The product's behaviors indicate important differences that may be observed by unauthorized actors in a way that reveals (1) its internal state or decision process, or (2) differences from other products with equivalent functionality.

Extended Description

Ideally, a product should provide as little information about its internal operations as possible. Otherwise, attackers could use knowledge of these internal operations to simplify or optimize their attack. In some cases, behavioral discrepancies can be used by attackers to form a side channel.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	203	Observable Discrepancy	518
ParentOf	V	206	Observable Internal Behavioral Discrepancy	527
ParentOf	V	207	Observable Behavioral Discrepancy With Equivalent Products	528
CanPrecede	C	514	Covert Channel	1218

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

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality Access Control	Read Application Data Bypass Protection Mechanism	

Observed Examples

Reference	Description
CVE-2002-0208	Product modifies TCP/IP stack and ICMP error messages in unusual ways that show the product is in use. *https://www.cve.org/CVERecord?id=CVE-2002-0208*
CVE-2004-2252	Behavioral infoleak by responding to SYN-FIN packets. *https://www.cve.org/CVERecord?id=CVE-2004-2252*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Behavioral Discrepancy Infoleak
WASC	45		Fingerprinting

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
541	Application Fingerprinting
580	System Footprinting

CWE-206: Observable Internal Behavioral Discrepancy

Weakness ID : 206
Structure : Simple
Abstraction : Variant

Description

The product performs multiple behaviors that are combined to produce a single result, but the individual behaviors are observable separately in a way that allows attackers to reveal internal state or internal decision points.

Extended Description

Ideally, a product should provide as little information as possible to an attacker. Any hints that the attacker may be making progress can then be used to simplify or optimize the attack. For example, in a login procedure that requires a username and password, ultimately there is only one decision: success or failure. However, internally, two separate actions are performed: determining if the username exists, and checking if the password is correct. If the product behaves differently based on whether the username exists or not, then the attacker only needs to concentrate on the password.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	205	Observable Behavioral Discrepancy	526

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Setup generic response pages for error conditions. The error page should not disclose information about the success or failure of a sensitive operation. For instance, the login page should not confirm that the login is correct and the password incorrect. The attacker who tries random account name may be able to guess some of them. Confirming that the account exists would make the login page more susceptible to brute force attack.

Observed Examples

Reference	Description
CVE-2002-2031	File existence via infoleak monitoring whether "onerror" handler fires or not. *https://www.cve.org/CVERecord?id=CVE-2002-2031*
CVE-2005-2025	Valid groupname enumeration via behavioral infoleak (sends response if valid, doesn't respond if not). *https://www.cve.org/CVERecord?id=CVE-2005-2025*
CVE-2001-1497	Behavioral infoleak in GUI allows attackers to distinguish between alphanumeric and non-alphanumeric characters in a password, thus reducing the search space. *https://www.cve.org/CVERecord?id=CVE-2001-1497*
CVE-2003-0190	Product immediately sends an error message when user does not exist instead of waiting until the password is provided, allowing username enumeration. *https://www.cve.org/CVERecord?id=CVE-2003-0190*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Internal behavioral inconsistency infoleak

CWE-207: Observable Behavioral Discrepancy With Equivalent Products

Weakness ID : 207
Structure : Simple
Abstraction : Variant

Description

The product operates in an environment in which its existence or specific identity should not be known, but it behaves differently than other products with equivalent functionality, in a way that is observable to an attacker.

Extended Description

For many kinds of products, multiple products may be available that perform the same functionality, such as a web server, network interface, or intrusion detection system. Attackers often perform "fingerprinting," which uses discrepancies in order to identify which specific product is in use. Once the specific product has been identified, the attacks can be made more customized and efficient. Often, an organization might intentionally allow the specific product to be identifiable. However, in some environments, the ability to identify a distinct product is unacceptable, and it is expected that every product would behave in exactly the same way. In these more restricted environments, a behavioral difference might pose an unacceptable risk if it makes it easier to identify the product's vendor, model, configuration, version, etc.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	205	Observable Behavioral Discrepancy	526

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality Access Control	Read Application Data Bypass Protection Mechanism	

Observed Examples

Reference	Description
CVE-2002-0208	Product modifies TCP/IP stack and ICMP error messages in unusual ways that show the product is in use. *https://www.cve.org/CVERecord?id=CVE-2002-0208*
CVE-2004-2252	Behavioral infoleak by responding to SYN-FIN packets. *https://www.cve.org/CVERecord?id=CVE-2004-2252*
CVE-2000-1142	Honeypot generates an error with a "pwd" command in a particular directory, allowing attacker to know they are in a honeypot system. *https://www.cve.org/CVERecord?id=CVE-2000-1142*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			External behavioral inconsistency infoleak

CWE-208: Observable Timing Discrepancy

Weakness ID : 208
Structure : Simple

Abstraction : Base

Description

Two separate operations in a product require different amounts of time to complete, in a way that is observable to an actor and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

Extended Description

In security-relevant contexts, even small variations in timing can be exploited by attackers to indirectly infer certain details about the product's internal operations. For example, in some cryptographic algorithms, attackers can use timing differences to infer certain properties about a private key, making the key easier to guess. Timing discrepancies effectively form a timing side channel.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

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

Nature	Type	ID	Name	Page
ChildOf	B	203	Observable Discrepancy	518
ParentOf	B	1254	Incorrect Comparison Logic Granularity	2060
CanPrecede	C	327	Use of a Broken or Risky Cryptographic Algorithm	799
CanPrecede	B	385	Covert Timing Channel	940

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1012	Cross Cutting	2427

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

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Applicable Platforms

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

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	
Access Control	Bypass Protection Mechanism	

Demonstrative Examples

Example 1:

Consider an example hardware module that checks a user-provided password to grant access to a user. The user-provided password is compared against a golden value in a byte-by-byte manner.

Example Language: Verilog *(Bad)*

```
always_comb @ (posedge clk)
begin
   assign check_pass[3:0] = 4'b0;
   for (i = 0; i < 4; i++) begin
      if (entered_pass[(i*8 - 1) : i] eq golden_pass([i*8 - 1) : i])
         assign check_pass[i] = 1;
         continue;
      else
         assign check_pass[i] = 0;
         break;
      end
   assign grant_access = (check_pass == 4'b1111) ? 1'b1: 1'b0;
end
```

Since the code breaks on an incorrect entry of password, an attacker can guess the correct password for that byte-check iteration with few repeat attempts.

To fix this weakness, either the comparison of the entire string should be done all at once, or the attacker is not given an indication whether pass or fail happened by allowing the comparison to run through all bits before the grant_access signal is set.

Example Language: *(Good)*

```
always_comb @ (posedge clk)
begin
   assign check_pass[3:0] = 4'b0;
   for (i = 0; i < 4; i++) begin
      if (entered_pass[(i*8 - 1) : i] eq golden_pass([i*8 -1) : i])
         assign check_pass[i] = 1;
         continue;
      else
         assign check_pass[i] = 0;
         continue;
      end
   assign grant_access = (check_pass == 4'b1111) ? 1'b1: 1'b0;
end
```

Example 2:

In this example, the attacker observes how long an authentication takes when the user types in the correct password.

When the attacker tries their own values, they can first try strings of various length. When they find a string of the right length, the computation will take a bit longer, because the for loop will run at least once. Additionally, with this code, the attacker can possibly learn one character of the password at a time, because when they guess the first character right, the computation will take longer than a wrong guesses. Such an attack can break even the most sophisticated password with a few hundred guesses.

Example Language: Python *(Bad)*

```
def validate_password(actual_pw, typed_pw):
   if len(actual_pw) <> len(typed_pw):
      return 0
   for i in len(actual_pw):
      if actual_pw[i] <> typed_pw[i]:
         return 0
   return 1
```

Note that in this example, the actual password must be handled in constant time as far as the attacker is concerned, even if the actual password is of an unusual length. This is one reason why it is good to use an algorithm that, among other things, stores a seeded cryptographic one-way hash of the password, then compare the hashes, which will always be of the same length.

Observed Examples

Reference	Description
CVE-2019-10071	Java-oriented framework compares HMAC signatures using String.equals() instead of a constant-time algorithm, causing timing discrepancies *https://www.cve.org/CVERecord?id=CVE-2019-10071*

Reference	Description
CVE-2019-10482	Smartphone OS uses comparison functions that are not in constant time, allowing side channels *https://www.cve.org/CVERecord?id=CVE-2019-10482*
CVE-2014-0984	Password-checking function in router terminates validation of a password entry when it encounters the first incorrect character, which allows remote attackers to obtain passwords via a brute-force attack that relies on timing differences in responses to incorrect password guesses, aka a timing side-channel attack. *https://www.cve.org/CVERecord?id=CVE-2014-0984*
CVE-2003-0078	SSL implementation does not perform a MAC computation if an incorrect block cipher padding is used, which causes an information leak (timing discrepancy) that may make it easier to launch cryptographic attacks that rely on distinguishing between padding and MAC verification errors, possibly leading to extraction of the original plaintext, aka the "Vaudenay timing attack." *https://www.cve.org/CVERecord?id=CVE-2003-0078*
CVE-2000-1117	Virtual machine allows malicious web site operators to determine the existence of files on the client by measuring delays in the execution of the getSystemResource method. *https://www.cve.org/CVERecord?id=CVE-2000-1117*
CVE-2003-0637	Product uses a shorter timeout for a non-existent user than a valid user, which makes it easier for remote attackers to guess usernames and conduct brute force password guessing. *https://www.cve.org/CVERecord?id=CVE-2003-0637*
CVE-2003-0190	Product immediately sends an error message when a user does not exist, which allows remote attackers to determine valid usernames via a timing attack. *https://www.cve.org/CVERecord?id=CVE-2003-0190*
CVE-2004-1602	FTP server responds in a different amount of time when a given username exists, which allows remote attackers to identify valid usernames by timing the server response. *https://www.cve.org/CVERecord?id=CVE-2004-1602*
CVE-2005-0918	Browser allows remote attackers to determine the existence of arbitrary files by setting the src property to the target filename and using Javascript to determine if the web page immediately stops loading, which indicates whether the file exists or not. *https://www.cve.org/CVERecord?id=CVE-2005-0918*

Functional Areas

- Cryptography
- Authentication

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	967	SFP Secondary Cluster: State Disclosure	888	2403
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Notes

Relationship

Often primary in cryptographic applications and algorithms.

Taxonomy Mappings