# CWE Detail – CWE-20

## Description

The product receives input or data, but it does
 not validate or incorrectly validates that the input has the
 properties that are required to process the data safely and
 correctly.

## Extended Description

Input validation is a frequently-used technique
 for checking potentially dangerous inputs in order to
 ensure that the inputs are safe for processing within the
 code, or when communicating with other components. Input can consist of: raw data - strings, numbers, parameters, file contents, etc. metadata - information about the raw data, such as headers or size Data can be simple or structured. Structured data
 can be composed of many nested layers, composed of
 combinations of metadata and raw data, with other simple or
 structured data. Many properties of raw data or metadata may need
 to be validated upon entry into the code, such
 as: specified quantities such as size, length, frequency, price, rate, number of operations, time, etc. implied or derived quantities, such as the actual size of a file instead of a specified size indexes, offsets, or positions into more complex data structures symbolic keys or other elements into hash tables, associative arrays, etc. well-formedness, i.e. syntactic correctness - compliance with expected syntax lexical token correctness - compliance with rules for what is treated as a token specified or derived type - the actual type of the input (or what the input appears to be) consistency - between individual data elements, between raw data and metadata, between references, etc. conformance to domain-specific rules, e.g. business logic equivalence - ensuring that equivalent inputs are treated the same authenticity, ownership, or other attestations about the input, e.g. a cryptographic signature to prove the source of the data Implied or derived properties of data must often
 be calculated or inferred by the code itself. Errors in
 deriving properties may be considered a contributing factor
 to improper input validation.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

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

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

**•** CVE-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. (KEV)

**•** 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. (KEV)

**•** CVE-2021-22205: Chain: backslash followed by a newline can bypass a validation step (CWE-20), leading to eval injection (CWE-95), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-21220: Chain: insufficient input validation (CWE-20) in browser allows heap corruption (CWE-787), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-9054: Chain: improper input validation (CWE-20) in username parameter, leading to OS command injection (CWE-78), as exploited in the wild per CISA KEV. (KEV)

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

**•** CVE-2020-3161: Improper input validation of HTTP requests in IP phone, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-3580: Chain: improper input validation (CWE-20) in firewall product leads to XSS (CWE-79), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-37147: Chain: caching proxy server has improper input validation (CWE-20) of headers, allowing HTTP response smuggling (CWE-444) using an "LF line ending"

**•** CVE-2008-5305: Eval injection in Perl program using an ID that should only contain hyphens and numbers.

**•** CVE-2008-2223: SQL injection through an ID that was supposed to be numeric.

**•** CVE-2008-3477: lack of input validation in spreadsheet program leads to buffer overflows, integer overflows, array index errors, and memory corruption.

**•** CVE-2008-3843: insufficient validation enables XSS

**•** CVE-2008-3174: driver in security product allows code execution due to insufficient validation

**•** CVE-2007-3409: infinite loop from DNS packet with a label that points to itself

**•** CVE-2006-6870: infinite loop from DNS packet with a label that points to itself

**•** CVE-2008-1303: missing parameter leads to crash

**•** CVE-2007-5893: HTTP request with missing protocol version number leads to crash

**•** CVE-2006-6658: request with missing parameters leads to information exposure

**•** CVE-2008-4114: system crash with offset value that is inconsistent with packet size

**•** CVE-2006-3790: size field that is inconsistent with packet size leads to buffer over-read

**•** CVE-2008-2309: product uses a denylist to identify potentially dangerous content, allowing attacker to bypass a warning

**•** CVE-2008-3494: security bypass via an extra header

**•** CVE-2008-3571: empty packet triggers reboot

**•** CVE-2006-5525: incomplete denylist allows SQL injection

**•** CVE-2008-1284: NUL byte in theme name causes directory traversal impact to be worse

**•** CVE-2008-0600: kernel does not validate an incoming pointer before dereferencing it

**•** CVE-2008-1738: anti-virus product has insufficient input validation of hooked SSDT functions, allowing code execution

**•** CVE-2008-1737: anti-virus product allows DoS via zero-length field

**•** CVE-2008-3464: driver does not validate input from userland to the kernel

**•** CVE-2008-2252: kernel does not validate parameters sent in from userland, allowing code execution

**•** CVE-2008-2374: lack of validation of string length fields allows memory consumption or buffer over-read

**•** CVE-2008-1440: lack of validation of length field leads to infinite loop

**•** CVE-2008-1625: lack of validation of input to an IOCTL allows code execution

**•** CVE-2008-3177: zero-length attachment causes crash

**•** CVE-2007-2442: zero-length input causes free of uninitialized pointer

**•** CVE-2008-5563: crash via a malformed frame structure

**•** CVE-2008-5285: infinite loop from a long SMTP request

**•** CVE-2008-3812: router crashes with a malformed packet

**•** CVE-2008-3680: packet with invalid version number leads to NULL pointer dereference

**•** CVE-2008-3660: crash via multiple "." characters in file extension

## Related Attack Patterns (CAPEC)

* CAPEC-10
* CAPEC-101
* CAPEC-104
* CAPEC-108
* CAPEC-109
* CAPEC-110
* CAPEC-120
* CAPEC-13
* CAPEC-135
* CAPEC-136
* CAPEC-14
* CAPEC-153
* CAPEC-182
* CAPEC-209
* CAPEC-22
* CAPEC-23
* CAPEC-230
* CAPEC-231
* CAPEC-24
* CAPEC-250
* CAPEC-261
* CAPEC-267
* CAPEC-28
* CAPEC-3
* CAPEC-31
* CAPEC-42
* CAPEC-43
* CAPEC-45
* CAPEC-46
* CAPEC-47
* CAPEC-473
* CAPEC-52
* CAPEC-53
* CAPEC-588
* CAPEC-63
* CAPEC-64
* CAPEC-664
* CAPEC-67
* CAPEC-7
* CAPEC-71
* CAPEC-72
* CAPEC-73
* CAPEC-78
* CAPEC-79
* CAPEC-8
* CAPEC-80
* CAPEC-81
* CAPEC-83
* CAPEC-85
* CAPEC-88
* CAPEC-9

## Attack TTPs

**•** T1574.007: Path Interception by PATH Environment Variable (Tactics: persistence, privilege-escalation, defense-evasion)

**•** T1539: Steal Web Session Cookie (Tactics: credential-access)

**•** T1553.002: Code Signing (Tactics: defense-evasion)

**•** T1574.006: Dynamic Linker Hijacking (Tactics: persistence, privilege-escalation, defense-evasion)

**•** T1562.003: Impair Command History Logging (Tactics: defense-evasion)

**•** T1027: Obfuscated Files or Information (Tactics: defense-evasion)

**•** T1036.001: Invalid Code Signature (Tactics: defense-evasion)

## Modes of Introduction

**•** Architecture and Design: N/A

**•** Implementation: REALIZATION: This weakness is caused during implementation of an architectural security tactic. If a programmer believes that an attacker cannot modify certain inputs, then the programmer might not perform any input validation at all. For example, in web applications, many programmers believe that cookies and hidden form fields can not be modified from a web browser (CWE-472), although they can be altered using a proxy or a custom program. In a client-server architecture, the programmer might assume that client-side security checks cannot be bypassed, even when a custom client could be written that skips those checks (CWE-602).

## Common Consequences

**•** Impact: DoS: Crash, Exit, or Restart, DoS: Resource Consumption (CPU), DoS: Resource Consumption (Memory) — Notes: An attacker could provide unexpected values and cause a program crash or arbitrary control of resource allocation, leading to excessive consumption of resources such as memory and CPU.

**•** Impact: Read Memory, Read Files or Directories — Notes: An attacker could read confidential data if they are able to control resource references.

**•** Impact: Modify Memory, Execute Unauthorized Code or Commands — Notes: An attacker could use malicious input to modify data or possibly alter control flow in unexpected ways, including arbitrary command execution.

## Potential Mitigations

**•** Architecture and Design: Consider using language-theoretic security (LangSec) techniques that characterize inputs using a formal language and build "recognizers" for that language. This effectively requires parsing to be a distinct layer that effectively enforces a boundary between raw input and internal data representations, instead of allowing parser code to be scattered throughout the program, where it could be subject to errors or inconsistencies that create weaknesses. [REF-1109] [REF-1110] [REF-1111] (Effectiveness: N/A)

**•** Architecture and Design: 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). (Effectiveness: N/A)

**•** Architecture and Design: Understand all the potential areas where untrusted inputs can enter the product, including but not limited to: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls. (Effectiveness: N/A)

**•** Implementation: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. (Effectiveness: High)

**•** 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. (Effectiveness: N/A)

**•** Implementation: When your application combines data from multiple sources, perform the validation after the sources have been combined. The individual data elements may pass the validation step but violate the intended restrictions after they have been combined. (Effectiveness: N/A)

**•** 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. (Effectiveness: N/A)

**•** Implementation: Directly convert your input type into the expected data type, such as using a conversion function that translates a string into a number. After converting to the expected data type, ensure that the input's values fall within the expected range of allowable values and that multi-field consistencies are maintained. (Effectiveness: N/A)

**•** Implementation: Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180, CWE-181). Make sure that your application does not inadvertently decode the same input twice (CWE-174). Such errors could be used to bypass allowlist schemes by introducing dangerous inputs after they have been checked. Use libraries such as the OWASP ESAPI Canonicalization control. Consider performing repeated canonicalization until your input does not change any more. This will avoid double-decoding and similar scenarios, but it might inadvertently modify inputs that are allowed to contain properly-encoded dangerous content. (Effectiveness: N/A)

**•** 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. (Effectiveness: N/A)

## Applicable Platforms

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

## Demonstrative Examples

**•** The user has no control over the price variable, however the code does not prevent a negative value from being specified for quantity. If an attacker were to provide a negative value, then the user would have their account credited instead of debited.

**•** While this code checks to make sure the user cannot specify large, positive integers and consume too much memory, it does not check for negative values supplied by the user. As a result, an attacker can perform a resource consumption (CWE-400) attack against this program by specifying two, large negative values that will not overflow, resulting in a very large memory allocation (CWE-789) and possibly a system crash. Alternatively, an attacker can provide very large negative values which will cause an integer overflow (CWE-190) and unexpected behavior will follow depending on how the values are treated in the remainder of the program.

**•** The programmer intended for $birthday to be in a date format and $homepage to be a valid URL. However, since the values are derived from an HTTP request, if an attacker can trick a victim into clicking a crafted URL with <script> tags providing the values for birthday and / or homepage, then the script will run on the client's browser when the web server echoes the content. Notice that even if the programmer were to defend the $birthday variable by restricting input to integers and dashes, it would still be possible for an attacker to provide a string of the form:

**•** 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.

**•** The application assumes the URL will always be included in the intent. When the URL is not present, the call to getStringExtra() will return null, thus causing a null pointer exception when length() is called.

## Notes

**•** 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. Multiple techniques exist to transform potentially dangerous input into something safe, which is different than "validation," which is a technique to check if an input is already safe. CWE users need to be cautious during root cause analysis to ensure that an issue is truly an input-validation problem.

**•** Maintenance: As of 2020, this entry is used more often than preferred, and it is a source of frequent confusion. It is being actively modified for CWE 4.1 and subsequent versions.

**•** Maintenance: Concepts such as validation, data transformation, and neutralization are being refined, so relationships between CWE-20 and other entries such as CWE-707 may change in future versions, along with an update to the Vulnerability Theory document.

**•** Maintenance: Input validation - whether missing or incorrect - is such an essential and widespread part of secure development that it is implicit in many different weaknesses. Traditionally, problems such as buffer overflows and XSS have been classified as input validation problems by many security professionals. However, input validation is not necessarily the only protection mechanism available for avoiding such problems, and in some cases it is not even sufficient. The CWE team has begun capturing these subtleties in chains within the Research Concepts view (CWE-1000), but more work is needed.

**•** Terminology: The "input validation" term is extremely common, but it is used in many different ways. In some cases its usage can obscure the real underlying weakness or otherwise hide chaining and composite relationships. Some people use "input validation" as a general term that covers many different neutralization techniques for ensuring that input is appropriate, such as filtering, i.e., attempting to remove dangerous inputs (related to CWE-790); encoding/escaping, i.e., attempting to ensure that the input is not misinterpreted when it is included in output to another component (related to CWE-116); or canonicalization, which often indirectly removes otherwise-dangerous inputs. Others use the term in a narrower context to simply mean "checking if an input conforms to expectations without changing it." CWE uses this narrow interpretation. Note that "input validation" has very different
 meanings to different people, or within different
 classification schemes. Caution must be used when
 referencing this CWE entry or mapping to it. For example,
 some weaknesses might involve inadvertently giving control
 to an attacker over an input when they should not be able
 to provide an input at all, but sometimes this is referred
 to as input validation. Finally, it is important to emphasize that the
 distinctions between input validation and output escaping
 are often blurred. Developers must be careful to
 understand the difference, including how input validation
 is not always sufficient to prevent vulnerabilities,
 especially when less stringent data types must be
 supported, such as free-form text. Consider a SQL injection
 scenario in which a person's 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, this valid name cannot be directly
 inserted into the database because it contains the "'"
 apostrophe character, which would need to be escaped or
 otherwise transformed. In this case, removing the
 apostrophe might reduce the risk of SQL injection, but it
 would produce incorrect behavior because the wrong name
 would be recorded.