Define an approach that ensures all data are explicitly validated

Software systems and components commonly make assumptions about data they operate on. It is important to explicitly ensure that such assumptions hold: Vulnerabilities frequently arise from implicit assumptions about data, which can be exploited if an attacker can subvert and invalidate these assumptions.

As such, it is important to design software systems to ensure that comprehensive data validation actually takes place and that all assumptions about data have been validated when they are used.

It is furthermore desirable to design software to make it feasible for a security reviewer to effectively and efficiently reason about and verify the correctness and comprehensiveness of data validation. Designing for verifiability should take into account that code typically evolves over time, resulting in the risk that gaps in data validation are introduced in later stages of the software life-cycle.

Design or use centralized validation mechanisms to ensure that all data entering a system (from the outside) or major component (from another component of the same system) are appropriately validated. For example:

- It is desirable for web applications to utilize a mechanism (such as a request filter or interceptor facility provided by the underlying web application framework) to centrally intercept all incoming requests, and to apply basic input validation to all request parameters.
- Implementations of communication protocols might centrally validate all fields of all received protocol messages before any actual processing takes place.
- Systems consuming complex data formats (such as XML documents, image file formats, or word processing file formats) might perform parsing, syntactic validation, and semantic validation of input files in a dedicated

Always consider the users

Understand how integrating external components changes your attack surface

Be flexible when considering future changes to objects and actors

Get Involved

validation module whose output is a validated internal object representation of the input document. Parsers and validators must themselves be designed to robustly cope with potentially malicious or malformed inputs.

Transform data into a canonical form, before performing actual syntactic or semantic validation. This ensures that validation cannot be bypassed by supplying inputs that are encoded in a transport encoding, or in a possibly invalid non-canonical form.

Use common libraries of validation primitives, such as predicates that recognize well-formed email addresses, URLs, and so forth. This ensures that all validation of different instances of the same type of data applies consistent validation semantics. Consistent use of common validation predicates can also increase the fidelity of static analysis. Validation should be based on a whitelisting approach, rather than blacklisting.

Input validation requirements are often state-dependent. For instance, in a stateful protocol, the set of valid values of a particular protocol message field (and hence the corresponding validation requirements) may depend on the protocol's state. In such scenarios, it can be beneficial to design the protocol implementation's input validation component to be itself state-aware.

Explicitly re-validate assumptions "nearby" code that relies on them. For example, the entry points of a web application's business-logic layer should explicitly re-state, and check as preconditions, all assumptions that it relies on. Liberal use of precondition checks in the entry points of software modules and components is highly recommended. Such precondition checks should never fail during execution of the deployed application, assuming the higher layers of the application have correctly validated external inputs. And as such, it is unnecessary for the business-logic layer to produce friendly error messages should such a precondition fail. Nevertheless, re-validation of data supplied to the business-logic layer provides two benefits:

It protects against vulnerabilities that arise from insufficient input validation in a higher layer (since the developer of the higher layer may not have a full understanding of all the requirements and assumptions of the lower layer), or from additional data-flows that were not considered during the initial security design (e.g., a data-load job that calls the business layer with data read from a file format used to exchange information between affiliated organizations, and which does not perform the same level of data validation as the web front end, based on the possibly invalid assumption that such files are "trusted").

- It permits local reasoning about the correctness of a component; since assumptions are explicitly checked and stated, a human reviewer or static analysis tool can truly assume the assumptions actually hold, without having to consider all (possibly very complex) data flows into the component.

Use implementation-language-level types to capture assumptions about data validity. For example, an application that receives as an input a date and time in string representation should validate that this input indeed consists of a well-formed string representation of a date and time (e.g., in ISO 8601 format). It is desirable to implement validation by parsing the input into a typed representation (such as a "date and time" type provided

in many programming language's standard libraries), and to use that typed representation (and not the original input string) throughout the program. Downstream components are then relieved from having to consider the possibility that a provided value (e.g., date) is syntactically invalid, and can focus on only checking additional preconditions that are not supported by the type's contract (e.g., that a date is not in the future).

Various problems arise from failure to address this security design principle.

- Injection vulnerabilities can arise if untrusted data are used without validation in certain contexts, such as APIs and platform features that process and interpret strings with certain semantics. For example:
  - Using an externally controlled string as a component of a file path can lead to path traversal vulnerabilities, unless the application validates that the input represents a single path component (and, in particular, does not contain path separators).
  - If an externally controlled string is used in a context in a HTML document where it will be interpreted as a URL, a Cross-Site Scripting (XSS) vulnerability can arise unless it has been validated that the string represents a well-formed URL with a benign scheme (such as http:/https:, and, in particular, not javascript:, vbscript:, data:, etc).

  It is generally preferable to perform data validation relevant to the prevention of injection vulnerabilities in the implementation of the API that is subject to injection vulnerabilities, or in a wrapper API in case the underlying API cannot be modified. See also "*Strictly separate data and control instructions, and never process control instructions received from untrusted sources*" section.
- Attempting to validate data that are not in canonical form can allow validation to be bypassed. For example, it is difficult to validate that an input string represents a single path component (free of path separator characters) unless the input has been fully decoded (with respect to transport encodings) and has been validated to be in a canonical character encoding — otherwise, it might be possible for an attacker to sneak a path separator past the input validation by representing it in an encoded form (e.g. %-encoding commonly used in web applications), or in the form of a non-canonical character encoding (e.g., a non-canonical UTF-8 encoding).
- In applications implemented in non-memory safe languages such as C, failing to carefully validate external inputs can result in memory corruption vulnerabilities such as buffer overflows, unbounded memory reads, null-terminated string issues, etc.
- Accepting inputs from untrusted sources without enforcement of an upper bound on data size can result in resource exhaustion.
- In general, aside from memory corruption and resource exhaustion issues, data that are not validated cause security issues primarily when they are used in a way that influences control flow. Data that are simply being copied around (e.g., received from an external input, then stored in a database, and later displayed in UI) are generally harmless. Problems arise if the application inspects the data and makes control flow decisions

based on the data's value. This most immediately applies to data that are used in contexts where they are interpreted as instructions or control, leading to injection vulnerabilities as discussed above. More generally however, control-flow dependencies on untrusted, non-validated data can lead to state corruption vulnerabilities, or execution of state transitions that the programmer did not intend or consider. Typically, security vulnerabilities in this category are highly domain- and application-specific, and hence are difficult to reason about and detect by general-purpose tools. Careful, state-dependent validation of inputs can go a long way towards mitigating this risk.