Strictly separate data and control instructions, and never process control instructions received from untrusted sources

Co-mingling data and control instructions in a single entity, especially a string, can lead to injection vulnerabilities. Lack of strict separation between data and code often leads to untrusted data controlling the execution flow of a software system. This is a general problem that manifests itself at several abstraction layers, from low-level machine instructions and hardware support to high-level virtual machine interpreters and application programming interfaces (APIs) that consume domain-specific language expressions.

At lower layers, lack of strict segregation between data and control instructions can result in the introduction of memory-corruption vulnerabilities, which in turn may permit attacker-controlled modifications of control flow or direct execution of attacker-controlled data as machine or byte-code instructions.

At higher levels, co-mingling of control and data often occurs in the context of runtime interpretation of both domain-specific and general-purpose programming languages. In many languages, control instructions and data are often segregated using in-band syntactic constructs, such as quoting and escaping. If software assembles a string in a parseable language by combining untrusted data with trusted control instructions, injection vulnerabilities arise if the untrusted data are insufficiently validated or escaped. In that situation, an attacker may be able to supply data crafted such that when the resulting expression is processed, parts of the data are parsed and interpreted as control (rather than uninterpreted data, as intended). Experience has shown that use of injection-prone APIs incurs significant risk that injection vulnerabilities will indeed be introduced. Examples of such vulnerabilities include SQL query injection, cross-site JavaScript injection, and shell command injection.

At lower levels, software platforms can utilize hardware capabilities to enforce separation of code and data. For example, memory access permissions can be used to mark memory that contains only data as non-executable

and to mark memory where code is stored as executable, but immutable, at runtime. Modern operating systems take advantage of such hardware features to implement security mechanisms that harden the entire software stack against multiple forms of attack. Software designs that ignore the principle of strict separation between data and code, or that blur the line that distinguishes one from the other, are inherently less secure because they undermine or directly invalidate low-level security mechanisms.

When designing languages, compilers, virtual machines, parsers and related pieces of infrastructure, consider control-flow integrity and segregation of control and potentially untrusted data as important design goals.

When designing APIs (both general-purpose or public interfaces as well as those that are domain- or application-specific), avoid exposing methods or endpoints that consume strings in languages that embed both control and data. Prefer instead to expose, for example, methods or endpoints that consume structured types that impose strict segregation between data and control information.

When designing applications that rely on existing APIs, avoid APIs that mingle data and control information in their parameters, especially when those parameters are strings. If there is no choice in underlying APIs (for example, if the use of a relational database requires interfacing through a SQL query API), it is often desirable to encapsulate the injection-prone interface and expose its functionality to application code through a higher-level API that enforces strict segregation between control statements and potentially untrusted data.

A design that relies on the ability to transform data into code should take special care to validate the data as fully as possible and to strictly constrain the set of computations that can be performed using data as an input language. Specific areas of concern include the eval function, query languages, and exposed reflection.

Eval Many interpreted languages (e.g. Python, Ruby, and JavaScript) have an

```
eval
```

function that consumes a string consisting of syntax in that language and invokes the language's interpreter on the string. Use of a language's

```
eval
```

facility can permit the implementation of very powerful features with little code, and is therefore tempting. It is also very dangerous. If attackers can influence even part of a string that is evaluated and that substring is not appropriately validated or encoded, they can often execute arbitrary code as a result.

Query languages Ensuring that appropriate validation or escaping is consistently applied in all code that interfaces with the query API is a difficult and error-prone process; implementing that functionality repeatedly

increases the risk of injection vulnerabilities. Use or develop an API that mediates between application code and raw query-language based interfaces (e.g., SQL, LDAP) and exposes a safer API. Avoid code that constructs queries based on ad-hoc string concatenation of fixed query stanzas with potentially untrusted data.

Exposed reflection Many programming languages provide facilities that allow programs to reflectively inspect and manipulate objects, as well as to invoke methods on objects. Use of reflection can be very powerful, and often permits the implementation of complex features using minimal code. For example, implementations of object serializers and deserializers used to marshal and unmarshal in-memory objects into and from a serialized form for persistence or network transfer can often be implemented very effectively using reflection.

However, as with

```
eval
```

, use of reflection can be a risky design choice. Unless inputs processed with reflection are very carefully controlled, bugs can arise that may permit the attacker to execute arbitrary code in the receiving process. It is often preferable to consider alternative, safer designs. For example, consider a design based on code-generation: A code-generated, reflection-free object serializer/deserializer is restricted to behaviors allowed by the explicitly generated code. This code is in turn generated at build/compile-time, where the code-generation process cannot be influenced by malicious inputs.