Unifying Models for Families of Common Weaknesses

Lambert Meertens[1]
meertens@kestrel.edu

Kestrel Institute Technical Note KES.U.11.01
February 2011

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
www.kestrel.edu

Abstract

This technical note presents a formal, semantic model for a class of common software vulnerabilities, identified in the Common Weakness Enumeration[2] as improper input validation[3]. The formalization is in terms of a failure of certain operations to commute.

Introduction

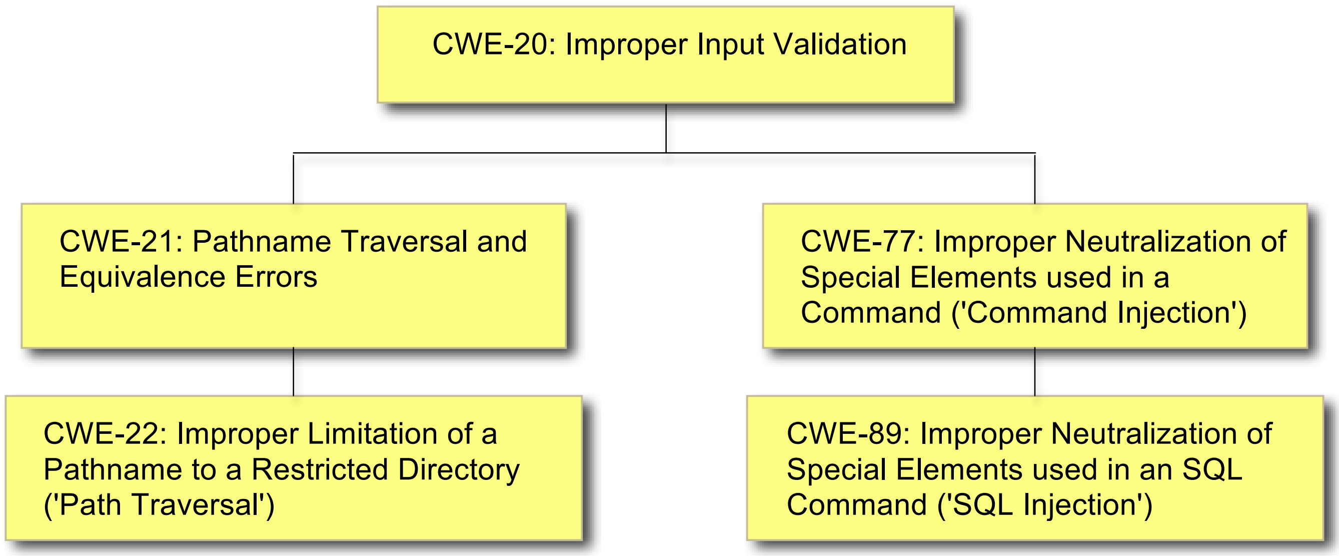
Many software attacks exploit vulnerabilities that have identifiable and well-understood types, as described in the Common Weakness Enumeration (CWE) repository maintained by MITRE. The purpose of this note is to point out the existence of quite specific and identifiable commonalities shared by some of the CWE weaknesses by defining a model from which different weaknesses can be obtained by instantiation.

For guarding against attacks exploiting software vulnerabilities, it is essential that the exploitable weaknesses be understood. Having precise models of such weaknesses allows to study them and to define measures that can prevent their introduction and facilitate their identification and removal from deployed software. More importantly, while the CWE repository lists specific types of weaknesses only identified after attacks based on them have been successfully mounted, models that unify exploitable weaknesses into a family allow for a pro-active approach that can identify new types of more specific weaknesses belonging to a family before they become actual vulnerabilities in deployed software.

[1] *Funded by AFRL contract #FA8750-11-C-0005.*
[2] http://cwe.mitre.org/
[3] CWE #20, which includes SQL injection, command injection and pathname traversal.

Example

To illustrate the concept, we define a model for a family of weaknesses that belong to class CWE-20 (Improper Input Validation), and show that two specific weaknesses, both listed in the 2010 CWE/SANS Top 25 Most Dangerous Software Errors, are instances. The figure below shows the taxonomy in the CWE repository of the weaknesses involved.


CWE-20: Improper Input Validation
CWE-21: Pathname Traversal and Equivalence Errors
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Theory

The essence of the family of weaknesses to be specified is a confusion between a more semantic domain of internal values and a more syntactic domain of representations. Denoting these two domains as $\mathcal{A}$ (for "abstract" values) and $\mathcal{C}$ (for "concrete" representations), there is an interpretation relation[4] J relating these two domains. Expressed symbolically[5], we have:

$$J : \mathcal{C} \to \mathcal{A} .$$

Domain $\mathcal{A}$ can be thought of as expressing intention, whereas $\mathcal{C}$ is used for communication with entities that are external to the software component proper. This includes both input (for example, supplied by a user) and output (for example, requests to an external application, or arguments for remote procedure calls or operating-system calls). Relation J is injective; that is, at most one internal value may be related to any given syntactic representation. Expressed symbolically:

$$(cJa \wedge cJa') \Rightarrow (a = a') .$$

As part of its regular operation, the software component receives external input and transforms it to external output. The transformation can be described at the abstract level of intentions as a relation:

$$T_{\mathcal{A}} : \mathcal{A} \to \mathcal{A} .$$

The transformation is implemented at the concrete level as a relation:

$$T_{\mathcal{C}} : \mathcal{C} \to \mathcal{C} .$$

These relations can be shown together in a diagram:

[4] The reasons for using a binary relation instead of a function are as follows. For concreteness, think of $\mathcal{A}$ as being the domain of numbers, and $\mathcal{C}$ as being the domain of strings (sequences of characters). A number such as 7 can have multiple concrete decimal representations as a string of characters, such as "`7`" and "`007`". Therefore the two domains cannot be related by a normal (single-valued) function $f : \mathcal{A} \to \mathcal{C}$. Conversely, a given string of characters, such as "`l07`", can fail to be the representation of a number. (In this example, the problem is that the first character is the letter "`l`" and not the digit "`1`".) This means that the two domains cannot be related by a normal (total) function $g : \mathcal{C} \to \mathcal{A}$ either. Similarly, the transformations described later may be partial and need to be modeled mathematically as binary injective relations.

[5] The notation $R : \mathcal{X} \to \mathcal{Y}$ is equivalent to $R \subseteq \mathcal{X} \times \mathcal{Y}$, and xRy is equivalent to $(x, y) \in R$.

$$\begin{tikzcd}
\mathcal{A} \arrow[r, "T_{\mathcal{A}}"] & \mathcal{A} \\
C \arrow[u, "J"] \arrow[r, "T_C"] & C \arrow[u, "J"']
\end{tikzcd}$$

Problems arise when the concrete implementation T_C does not properly capture the intention. Specifically, let c be some element of C received as external input. If it is related to external output c' by T_C (that is, cT_Cc' holds), then c' must be related by J to some internal interpretation a' related by $T_{\mathcal{A}}$ to the internal interpretation a of c, that is, the unique value related by J to c.

This mouthful can be expressed symbolically in one line:

$$cT_Cc' \;\Rightarrow\; (\exists a',a : c'Ja' \wedge aT_{\mathcal{A}}a' \wedge cJa)\ .$$

Using standard operations of the relational calculus, an even more concise formula is given by:

$$T_C \subseteq J^{-1} \circ T_{\mathcal{A}} \circ J\ .$$

In the above diagram, the lower, direct path from C to C must be contained in the upper path leading from C to C, going against the direction of the arrow for the last step from $\mathcal{A}$ to C.

If all is well, c , c', a and a' are related as indicated below:

$$\begin{tikzcd}
a \arrow[r, "T_{\mathcal{A}}"] & a' \\
c \arrow[u, "J"] \arrow[r, "T_C"] & c' \arrow[u, "J"']
\end{tikzcd}$$

There are several ways in which the required containment could be violated. If, for given c and c', there is no a' such that $c'Ja'$ holds, this means that c' is invalid as input to the external entity it is intended for. If that entity properly validates its input and reports any issues detected, and the software component under scrutiny properly deals with such reports, there is no immediate vulnerability, although relying on external entities to guard against security risks is generally not robust and ill-advised. It is even possible that there is no internal interpretation a of c, which ought to be detected and reported back to the entity supplying input c, signalling that the input could not be interpreted.

If, however, there are interpretations a' and a such that both $c'Ja'$ and cJa hold, but containment is violated, this means there is a second interpretation a'', related to a by $T_{\mathcal{A}}$, that differs from a'. (For simplicity we assume that $T_{\mathcal{A}}$ is a total relation, that is, given $a \in \mathcal{A}$ there is some value a'' such that $a'' T_{\mathcal{A}} a$ holds.) In words, the external output to which the internal input is transformed does not reflect the intention of the transformation. Shown in a picture, we have:

$$\begin{tikzcd} a \arrow[r, "T_{\mathcal{A}}"] & a'' & \\ & a' & \text{mismatch} \\ c \arrow[uu, "J"] \arrow[r, "T_C"] & c' \arrow[u, "J"'] & \end{tikzcd}$$

This may create a vulnerability. A malicious attacker could possibly exploit this by creating input that gets transformed to output that does not fit within the parameters determined by security considerations. A security risk even exists if no malice is involved: harmful effects might also arise unintentionally.

Unifying model

In this section we model the theory as a Metaslang[6] specification:

```
ACmatch = spec
  import /Library/General/Relation

  type A
  type C

  op J   : Relation(C, A)

  op T_A : Relation(A, A)
  op T_C : Relation(C, C)

  axiom A_C_match is
    T_C <= inverse(J) o T_A o J

end
```

This specification closely follows the presentation of the theory given above. The axiom `A_C_match` specifies the requirement that the implementation T_C captures the intention of the transformation $T_{\mathcal{A}}$.

[6] Metaslang is the specification language of Kestrel Institute's Specware system.

Identifying potential weaknesses

To identify potential weaknesses in existing programs requires determining data-flow paths where information flows from (untrusted) external input channels to (unhardened) external output channels (taint propagation). This can be done automatically, using well-understood and standard data-flow analysis techniques, and the concrete transformation T_C the data is subjected to can automatically be extracted from the program and expressed as a relation.[7]

It further requires defining the relevant interpretation relations and intentional transformations, that is, relations J and $T_{\mathcal{A}}$. This can, in general, not be automated.

CWE-22 ('Path Traversal') as an instance

The domain involved in CWE-22 is that of a pathname (or, for short, "path") in a file system. In many systems (e.g. Windows and Unix/Linux) a path consists of a list of path segments, corresponding to the hierarchical structure of the file system. A path defines an access to a resource, ususally a file or a folder. Paths may be relative, starting from a given access point, or absolute, starting from the root of a file system. The syntactic notation may vary between file systems, such as Windows' `\projects\RCP\reports` vs. Unix' `/projects/RCP/reports`, but to simplify the presentation we abstract from the syntactic details. Examples will employ forward slashes.

Path segments are strings such as `projects` and `RCP`. There are two strings that have a special significance, the so-called "dot segments" consisting of one or two dots: `.` and `..`. Before a path is used to access a resource, it has to be normalized; a full (absolute) normalized path no longer contains any dot segments. The details are not relevant to the present exposition, but essentially this comes down to two rules, which are applied left-to-right:

1. Any dot segment of the form `.` is elided.
2. Any dot segment of the form `..` is elided, together with the preceding segment (if any)

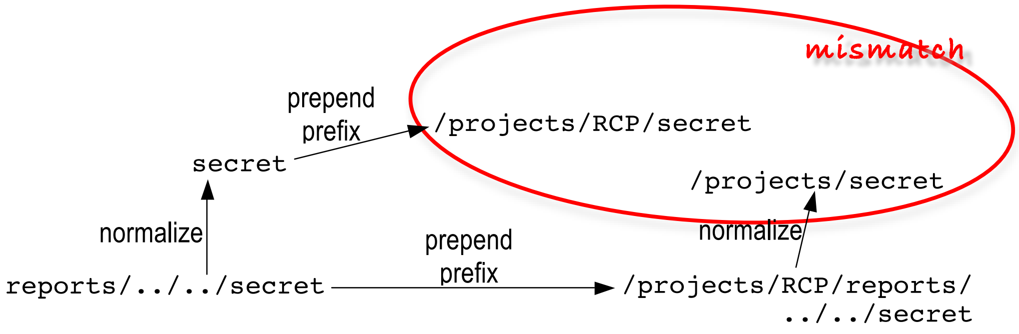
For example, `aa/./bb/cc/../dd` is thereby normalized to `aa/bb/dd`.

The transformation in the application forms an absolute path from a relative path supplied as input by prepending a given absolute path *prefixPath* that forms the top-level starting point in the hierarchy for this application for accessing

[7] The full generality of relations may not be needed in many cases, but there is a completely standard way of embedding functions in the world of relations, namely by mapping function f to the relation R_f such that $x\,R_f\,y$ holds precisely when $y = f(x)$.

requested resources, all of which have a path of which *prefixPath* is a prefix. At the intentional level, this prefix is prepended to the intentional interpretation of the input path supplied, that is, the path obtained by normalization.

The usual file-system facilities will already normalize any path presented as an argument of a system call, so it may seem that for the transformation at the concrete level no normalization is needed, and that just prepending the prefix path – assumed to be normalized itself – should be fine. In other words, it is tempting to assume that the effect of prepending a prefix followed by normalization is the same as the effect of normalization followed by prepending a prefix. However, it is possible to construct input for which this is not the case: the following example shows how a resource outside the project space set aside for project RCP could become exposed and accessible to an external agent.



It should be clear that CWE-22 is an instance of the general model for this family of weaknesses.

CWE-89 ('SQL Injection') as an instance

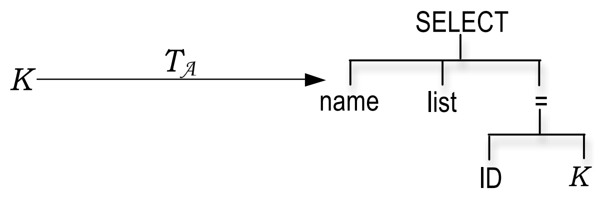
While in Path Traversal the abstract domain $\mathcal{A}$ and the concrete domain $\mathcal{C}$ were both taken to be paths, in modeling SQL Injection it is more profitable to use text strings for the concrete domain and SQL syntax trees for the abstract domain of intentions. The interpretation relation is given by the parsing relation that maps strings, provided they are valid SQL, to SQL syntax trees.

As an example we take a simple case in which an input key K is transformed to the SQL query The input key is a string literal, which stands for itself, so at the input side the interpretation relation is simply the identity. This is to be transformed into the SQL query

"`SELECT name FROM list WHERE ID = '`K`'`",

which shoud select the `name` field from relation `list` for the tuples for which the `ID` field is equal to the user-supplied input key.

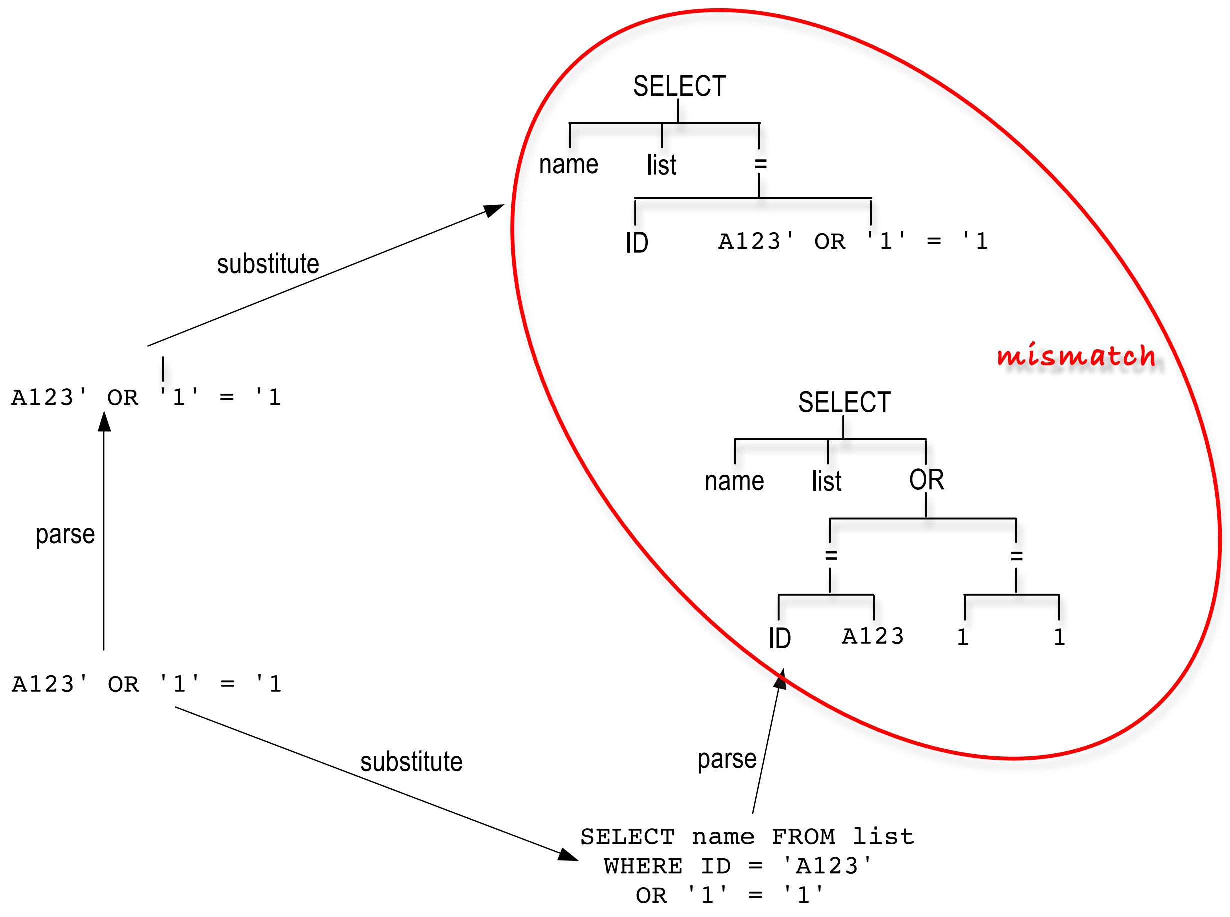
The intentional transformation is as in the following diagram:



Suppose further that the concrete implementation has been realized by naively substituting the input key textually in the query:

$$K \xrightarrow{T_C} \begin{array}{l}\texttt{SELECT name FROM list} \\ \quad\texttt{WHERE ID = '}K\texttt{'}\end{array}$$

Then if the key is, for example, "`A123' OR '1' = '1`", we get the following mismatch:



Literature

[Kes09] Kestrel Institute. *Specware Language Manual* and *Specware User Manual*. 2009. Online at www.specware.org.

[Khe09] Khedker, Uday, Amitabha Sanyal and Bageshri Karkare. *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

[Mee76] Meertens, Lambert. “From Abstract Variable to Concrete Representation”. In S. A. Schuman, editor, *New Directions in Algorithmic Languages 1976*, pp. 107–133, 1977.

[MIT10] MITRE. “2010 CWE/SANS Top 25 Most Dangerous Software Errors”. Online at cwe.mitre.org/top25.

Appendix

Details:

```
ACmatch = spec
  import /Library/General/Relation

  type A
  type C

  op J   : Relation(C, A)

  op T_A : Relation(A, A)
  op T_C : Relation(C, C)

  axiom A_C_match is
    T_C <= inverse(J) o T_A o J

end

ProgramFragment = spec
  import /Library/Base/String
  import /Library/Base/List

  type PathSegment = String
  type Path = List PathSegment

  op prefixPath : Path

  op  buildFilePath : Path -> Path
  def buildFilePath p = prefixPath ++ p

end

ProgramFragmentRel = spec
  import /Library/General/Relation
  import ProgramFragment

  op [a,b] f2R : (a -> b) -> Relation(a, b)
  def      f2R f (x, y) = (f x = y)

  op  normalizePath : Path -> Path
  def normalizePath = ...

  op  normalizePathRel : Relation(Path, Path)
  def normalizePathRel = f2R normalizePath
```

```
      op  buildFilePathRel : Relation(Path, Path)
      def buildFilePathRel = f2R buildFilePath

    end

    CWE22_safe = morphism ACmatch -> ProgramFragmentRel {
      A   +-> Path,
      C   +-> Path,
      J   +-> normalizePathRel,
      T_A +-> buildFilePathRel,
      T_C +-> buildFilePathRel
    }

    CWE22_obligations = obligations CWE22_safe
```

The obligations are the proof obligations engendered by the claim that the name translation specified in `CWE22_safe` results in a morphism from the `ACmatch` model to the relational description of the program fragment. Using the Specware `show` command they can be made visible:

```
    CWE22_obligations = spec
      import ProgramFragmentRel

      conjecture A_C_match is
        buildFilePathRel <= inverse(normalizePathRel)
                            o buildFilePathRel
                            o normalizePathRel

    end
```

There is just a single proof obligation, namely the obligation to prove the stated conjecture. Even assuming an axiom that `prefixPath` itself is already in normalized form, this will fail, thereby raising a red flag.