The VerSec Trust Schema Compiler

Trust schemas enforce security for the Data-Centric Toolkit (DCT). The VerSec Domain Specific Language describes a trust schema's rules. The `schemaCompile` compiler translates this description into a binary trust schema used by the `schemaLib` runtime to construct and validate DCT *Publications*. The language provides a simple, general framework for constructing self-consistent, validatable names. This directory contains schemaCompile and examples of its use in DCT. Applications using DCT each define Publication format(s) that are intimately related to the trust schema which lays out the rules for creating Publications and the associated signing chain(s).

Schema Description Language

The schema description language describes *constraints* on both the layout and components of names and on the structural and signing relationships between names. It is a referentially transparent, *declarative language* like Prolog or Datalog, based on Unification and Resolution. Its statements simply state facts that can be given in any order. Only when it has all the facts does the compiler analyze them to determine if they're consistent, complete and verifiable. Like Datalog, the language is deliberately not Turing complete. It's built using first-order logic fragements guaranteed to be decidable and have O(1) (constant time and space) validation and construction complexity. It also follows LangSec principles to minimize misconfiguration and attack surface. For example, the language can only produce entities (publications, certificates, signing chains, etc.) with a fixed layout so the run-time validatation is loop- and recursion-free. All alternatives (e.g., different signing chains for the same publication) must be explicitly enumerated so validation and construction are constant time and space. The language is intended to secure relatively local, mission-focused security domains (like a home or small business IoT network) where policy validity can be adjudicated via a common, local trust anchor. This means the set of signing chains for any schema must be a DAG. This property is verified at compile time and exploited to make the runtime testing for signing-loops stateless and O(1).

Lexical building blocks

Names consist of a sequence of *components* separated by slashes `/`

Each *component* is an *expression* that can be:

- a literal string enclosed in quotes `"`
- an *identifier*
- an internal function *call*
- an *expression* enclosed in parens
- two expressions separated by a vertical bar `|`.

Identifiers are a sequence of letters, digits and underscores starting with a letter, underscore or sharp sign. A leading underscore indicates that the component's value will be derived from schema rules and it *must* resolve to a concrete value (literal string, function call or signing chain component name) for the schema to be correct. Identifiers that don't start with '_' are *parameter*s that must be supplied at run time.

Comments start with "//" and terminate at the end-of-line. Comments, blank lines and whitespace are ignored.

Syntax and Semantics

A schema consists of a series of *statements*. A statement can describe either a *definition* or a *signing chain*. Statements must be terminated by a comma or newline.[1]

Definitions

A *definition* consists of an *identifier* followed by a colon followed by an *expression*. Semantically it means that replacing the identifier with the expression, or vice-versa, doesn't change the meaning of the schema.

The *expression* must be a *Name* (sequence of components separated by slashes) or another definition's identifier. It can be followed by (optional) *component constraints* and/or *signing constraints*. For example,

```
#mypub: /_domain/_topic/param
req: #mypub & {_topic: "request"}
```

defines `#mypub` as a three component name whose components can be referenced as '_domain', '_topic' and 'param' and `req` as a specialization of it whose *_topic* component is constrained to be the literal string "request".

In *Publication definitions*, like `#mypub` above, the expression must be a *Name* which is used to define the pub's *component tag identifiers.* Publication variant definitions, like `req` above, normally start with the parent pub's identifier followed by constraints to specialize it.

Component Constraints

A *component constraint* is an open brace followed by one or more constraint terms followed by a closing brace. Each constraint term consists of a *tag identifier* followed by a colon followed by an expression followed by a comma or newline. The semantics of a component constraint is that *all* its terms must hold. E.g.,

```
#mypub & {_topic: "req", param: "status"} | {_topic: "cmd", param:
"start"}
```

resolves to `/_domain/"req"/"status"` and `/_domain/"cmd"/"start"` while

```
#mypub & {_topic: "req"|"cmd", param: "status"|"start"}
```

resolves to the full cross product {req,cmd} x {status,start}.

An ampersand (&) must separate the component constraints from the definition's initial *Name* or *identifier*. As the examples show, multiple component constraints can be given, separated by `|` or `&`. The two operators have equal precedence so if a mixture of `|` and `&` is used, parens are needed to ensure the intended evaluation order.

Signing Constraints

A *signing constraint* consists of a `<=` (signed-by operator) followed by one or more *definition identifiers* separated by `|` operators. E.g.,

```
cmd: #mypub & {_topic: "cmd"} <= opCert
req: #mypub & {_topic: "req"} <= opCert | userCert
```

says that `cmd` publications must be signed by an `opCert` while `req` publications can be signed by either an `opCert` or a `userCert`. If the domain signing authority only signs an `opCert` for designated operators, the semantic enforced here by the run-time library is "operators can issue requests and commands but normal users can only issue requests". This is enforced *at the subscriber(s)* via their validating that `cmds` are signed by an `opcert`, so a compromised publisher forging a `cmd` or signing one with a `userCert` produces at most an "attempted security breach" log message. (A compromised publisher with access to a valid `opcert` is still a threat but that needs to be addressed by key hygiene measures like secure signing enclaves and limited key lifetimes.)

Signing Chains

A *signing chain* is a sequence of *identifiers* separated by `<=` operators. All of the identifiers must be defined as pubs or certs somewhere in the schema. The chain doesn't have to include all the certs on the path from the publication to the trust root --- *signing chain* and *signing constraint* information is interpreted as edges in a *signing DAG* that defines all the legal chains for the schema. (For diagnostic purposes the DAG is output in graphviz dot format in the compiler debug output.) The DAG is checked to insure:

- all nodes in the graph have been defined
- the cert graph is a DAG (there are no cycles)
- there is only one trust anchor (the DAG has a single sink)
- all signed publications have path(s) to the trust anchor

A publication's entire signing chain, not just its immediate signing cert, is used to establish provenance, authorizations, capabilities and trust. If a chain has variants that are not distinguished by the immediate signer, enough of the chain should be specified to disambiguate.

Syntax and Semantics

A *Publication* is the first element of each signing chain. The remaining elements must all be *Certificates*, each of which cryptographically signs the element preceding it. The final element must be a self-signed *Trust Anchor* so it signs both the preceding element and itself. This validation structure allows not just the publication's components but all the components in the chain to be used to enforce policy when building and validating publications. This is normally done implicitly via signing rules as in previous cmd/req example. But it can also be done explicitly by mirroring components within a cert chain. For example, for forensic logging it could be useful to know who issued the `cmd` and `req` publications of the previous example. Say `opCert` and `userCert` are defined as `roleCert` instances and `roleCert` is defined with a `_role` component containing the type of role and a `_roleID` component containing a person identifier like an employee number or name:

```
roleCert: /_domain/_role/_roleID
opCert: roleCert & {_role: "operator"}
userCert: roleCert & {_role: "user"}
```

then adding a `roleID` component to `#mypub`:

```
#mypub: /_domain/_topic/_roleID/param
```

will cause the runtime to set that component to the value of `_roleID` in the signing cert when the publication is built and verify the component matches the signing cert's when the publication is received. For non-parameter identifiers like `_roleID,` the compiler creates runtime schema *component correspondence* information that allows both the builder and verifier to ensure that *all* signing chain components with that identifier contain the same value.

Parameters and *correspondences* are essentially schema 'variables' that are bound to concrete values by the runtime to create a *logically grounded* (fully specified and verifiable) publication. To ensure that all generated pubs are grounded, the runtime schema describes all the *parameters* needed and the pub builder method throws an error if any of them are not supplied. For *correspondences,* the compiler checks that all non-parameter identifiers in a publication also appear in at least one cert of the pub's signing chain(s) and terminates with an error if not. (If no cert in the chain contains the identifier, that pub component can't be filled in with a concrete value at runtime so the schema doesn't produce grounded pubs and is in error.) Certs don't need this check because, at runtime, certs are pre-built, grounded objects and the runtime only has to check, once, that they meet the constraints of the schema.

Using the Schema Compiler

schemaCompile CLI

`schemaCompile [-v|-d|-q] [-o bschema] schemaInputFile`

Output verbosity control

There are currently five levels of output verbosity:

level	verbosity
0	no output
1	minimum information needed to use the runtime API: Gives the name, parameters and tags of each publication and templates for each cert.
2	adds each publication's signing chains
3	adds the information used to build and validate each publication (templates, signers, discriminators and correspondences)
4	adds internal state information (the cert DAG, token table and binary schema sections)

The compiler's default verbosity is 1. Each `-v` flag increments the level, `-d` sets it to max and `-q` sets it to zero.

Other flags

`-o file` dumps the binary schema to 'file' (for debugging)

`-p` turns on (voluminous) parser debug output (see the bison manual for details)

[1] Actually, all statements must be terminated by a comma but, as in javascript, the scanner will automatically insert a line terminator before a newline if the statement is complete (doesn't end with a binary operator, "{" or "("). ↩