Toward Provenance-Based Security for Configuration Languages

Paul Anderson and James Cheney
University of Edinburgh
dcspaul@ed.ac.uk, jcheney@inf.ed.ac.uk

Abstract

Large system installations are increasingly configured using high-level, mostly-declarative languages. Often, different users contribute data that is compiled centrally and distributed to individual systems. Although the systems themselves have been developed with reliability and availability in mind, the configuration compilation process can lead to unforeseen vulnerabilities because of the lack of access control on the different components combined to build the final configuration. Even if simple change-based access controls are applied to validate changes to the final version, changes can be lost or incorrectly attributed. Based on the growing literature on provenance for database queries and other models of computation, we identify a potential application area for provenance to securing configuration languages.

1 Introduction

In a computing infrastructure, *system configuration* [3] is the task of assigning configuration parameters to all of the individual components so that the overall system behaves according to requirements. The *infrastructure* may be a datacentre, or a distributed cloud application, or any other composition of connected systems. Typically, there will be many thousands of parameters which control the behaviour of the individual systems, and the relationships between them.

To manage this complexity, special-purpose system-configuration languages have evolved, such as LCFG [4, 1] and Puppet [9, 2]. These allow the configuration requirements to be specified in a relatively high level way, and the languages are compiled down to generate the individual configuration parameters. The system configuration task can then be thought of as "Programming the Infrastructure" [5], with the configuration parameters analogous to the machine code, and the configuration specifications analogous to a high-level program.

However, configuration languages are significantly different from programming languages. Most modern configuration languages adopt a (more or less) declarative approach to specifying the desired configuration. The configuration tool is then responsible for generating and sequencing the actions necessary to transform the current configuration into the desired one. This decouples the description of the configuration from the process of deploying it, and the configuration language becomes purely a data description language.

Because of the complexity, manual configuration of large infrastructures has become largely impractical, and configuration tools are now ubiquitous. However, these tools and languages are in their infancy. Unlike the programming languages used in mission-critical applications, there has been very little attempt to formalise the semantics, or verify their implementation. This can lead to a substantial discrepancy between the rigour of an application, and the infrastructure on which it depends:

> "In his case study about Linux system engineering in air traffic control, Stefan Schimanski showed how scalable Puppet really is and how it can guarantee reliable mass deployment of the Linux-based, mission critical applications needed in air traffic control centers."[1].

One specific problem is the lack of ability to track the provenance of particular configuration parameters. Large configurations are collaborative endeavours, involving many authors, and often spread across different organisations, so separation of concerns is an important consideration, and it is typical to see a hierarchy of classes with value-inheritance (rather than type inheritance). This allows a chain of users to successively specialise descriptions provided by others (see section 2 for an example). In practice, it can be extremely difficult to determine the sections of code, and the corresponding people who have contributed to the final value.

[1] FOSDEM 2011 (`http://lwn.net/Articles/428207/`)

Understanding the provenance of the resulting configuration parameters is important simply for managing the configuration — in the case of configuration errors, for example, it is clearly essential to understand who has contributed to a parameter, and in what way. However, the ability to identify the provenance of each parameter is also prerequisite for creating a more secure tool which supports a fine-grained authorisation of the configuration parameters. This type of authorisation is not supported by any of the existing high-level, declarative tools, and it is difficult to prevent anyone with access to the configuration source from modifying any aspect of the entire system.

This paper is intended to introduce system configuration languages as a potentially new and important application area for provenance, particularly with regard to securing and auditing changes to configurations. We provide some examples of configuration problems which could be avoided (or at least mitigated) by developing provenance-aware configuration languages and access control mechanisms. We highlight the lack of formal semantics for existing languages as a significant problem, and identify what we believe are promising next steps to take in solving these problems.

2 An Example

This section shows a (greatly simplified) example of the way in which large configurations are structured, and the kind of problems which can occur in practice:

1. **Alice** works for the company that supplies the configuration tool, and she develops generic templates, including one for a typical server machine. This has dozens of fields, including one for the "timeserver" which is the address of the machine against which to synchronise the operating system time. By default, this is set to some reliable, well-known public service:

   ```
   class genericServer {
       timeServer = ts@reliable.com
       ...
   }
   ```

2. **Bob** is the senior sysadmin for widgets.com and he develops templates for use in the company. These inherit values from the distributed templates. Bob overrides some of the default parameters, but leaves many at their default value — in particular, the timeserver default seems reasonable and he doesn't change it.

   ```
   class widgetServer isa genericServer {
      ...
   }
   ```

3. **Carol** is the sysadmin for the sales department and she inherits Bob's templates, again overriding some values to localise them, but leaving most (including the timeServer) with their defaults:

   ```
   class salesServer isa widgetServer {
      ...
   }
   ```

4. **Dave** is the technician who configures the individual machines. He just assigns one of Carol's templates to the machine and overrides a few host-specific parameters such as the machine name and address:

   ```
   class serverA isa salesServer {
      ip = 1.2.3.4
      ...
   }
   ```

   Carol carefully checks the new machine (`serverA`) and verifies that everything is ok. In particular, the timeServer has the reasonable value `ts@reliable.com`.

   Of course, in practice, there will be several thousand parameters, possibly hundreds or thousands of machines, and probably hundreds of classes. So this structure is designed to support a separation of concerns. For example, Dave can install the machine just by supplying the IP address and it gets installed with the policy defined by Carol. Carol only worries about departmental policy, assuming that Bob has taken care of company-level policy, etc.

5. At some point, Carol wants to experiment with a local timeserver. So, she might change her template to:

   ```
   class salesServer isa widgetServer {
      timeServer = ts@sales.widget.com
      ...
   }
   ```

   Nobody else needs to be concerned with this, and all the sales machines will now switch to using the new timeserver.

6. At some later point, Alice ships a new version of the generic templates. These define a new timeserver for some reason — perhaps a mistake, perhaps a malicious intervention, or perhaps something which is appropriate for many companies — although not for widget.com:

   ```
   class genericServer {
      timeServer = ts@unreliable.com
      ...
   }
   ```

The generic templates are too large to inspect in detail, but Bob is careful. Before he makes them live, he performs regression testing by doing a test compilation and inspecting the values which have changed in all the machines, such as `serverA`. Some things may have changed on Carol's machines (and presumably they are acceptable), but the timeserver will not have changed (because of the local override) and won't appear to be a problem. So the new templates are made live.

7. At some later time, Carol decides to withdraw her local timeserver, so she removes the override:

```
class salesServer isa widgetServer {
   ...
}
```

   At this point, all of the machines in the sales department inherit the unreliable server. The users blame Dave who says he hasn't changed anything. Carol says she just put the configuration back to the default, so it can't be her fault. Bob says he hasn't changed anything for months, so it can't be his fault! In a large system, it may not be obvious how the machines have ended up with the unreliable value, and where it came from. This takes a good deal of human communication to sort out, defeating the separation of concerns.

Most configuration languages would allow us to add validation to the resource values, so someone (probably Bob) could specify that the timeserver must have some particular property (perhaps being a member of specific domains). But the validation can only take place on the final instantiated values. So this prevents the bad configurations being deployed, but it still means that the problem is not detected until later, and the root cause is no clearer. Not only would Carol be prevented from deploying her change, but no further configuration changes could be deployed until this had been resolved (or the unwanted override re-instated).

3 Access control

Perhaps it was not appropriate for Carol to change the timeserver at all (company policy, perhaps). In this case, a fine-grained access control would have allowed Bob to prevent Carol from setting this value. As noted, such control requires a clear notion of provenance though. Vanbrabant et al. [10] proposed a solution to this which involves analysing the differences between successive configurations, in the context of Puppet [2], a popular, open-source configuration language. However, in the absence of provenance information, this mechanism attributes all of the changed values to the user initiating the change. In our example, Carol would have been blamed for the failure, although she did not author the offending resource, nor was she the appropriate person to fix the problem.

We suspect that computing provenance for real configuration languages such as Puppet will present significant challenges; we have already noted that the semantics (and even the syntax) is not clear, and certainly not formal. It is also evolving with reasonably frequent ad-hoc additions and changes. However, it is also likely that there are language constructs which make it particularly difficult to derive a useful provenance. For example, the inheritance operation illustrated above has a clearly corresponding provenance derivation (left projection). There may be other composition operators where the corresponding provenance derivation is simply a union — for example appending two lists, or adding two numbers — in such cases the final provenance for a value might simply say "everyone was responsible for this in some way".

It is possible for a configuration to produce acceptable parameters, but still be suspect in some way. For example, there is something fragile about the configuration resulting from stage 6. It now depends on Carol's override value for its correctness — which was not previously the case.

As another example, suppose (after untangling the above mess), Bob sets the timeserver field of widgetServer to `ts@widget.com` and also installs fine-grained access control rules that say that only Bob can set the timeserver in any `widgetServer`. Carol, independently, sets the timeserver in `salesServer` to `ts@widget.com`, inadvertently duplicating Bob's setting. This is fine, because Carol's change confirms Bob's setting, so there is no visible change in the end result for the access control checker to complain about. However, later the timeserver needs to be upgraded and Bob tries to set the `widgetServer.timeServer` to `ts2@widget.com`. Carol's change now overrides Bob's, and the final configuration does not change, so the change is allowed but has no effect, and Bob does not notice this. The next day, when the main server is switched off, everything grinds to a halt, and Carol tries to manually set the time server to a local machine to get the sales machines running again. However, since this change does affect the final time server value, the change is disallowed since it is not made by Bob. Recovering from this inconsistent state requires Bob to either deactivate access control or revert to a consistent configuration and and coordinate with Carol.

It is worth pointing out that part of the problem here is the distributed way that configurations are managed: the files Alice, Bob, Carol and Dave are editing are all likely to be versioned, but may be hosted on different systems

under no central control. Centralizing this information in a database that maintains the final version of the configuration, and controlling all users' access to this information, might help make it easier to detect bad changes earlier, but this mode of operation would be far less appealing and convenient than the conventional file-based approach: users would have to formulate bulk changes in terms of queries and updates, and all users (including Alice, not an employee of widget.com) would have to have accounts on the database.

Thus, we seek an alternative solution that preserves the good properties of the conventional distributed approach to configuration while also addressing the problems discussed above. These problems could be identified earlier if we could specify finer-grained policies controlling the changes made by intermediate users, not just the end result.

In the rest of the paper, we suggest ways that provenance techniques developed in the context of databases could be adapted to this problem.

4 Next steps and open questions

Where-provenance [7, 6]. As a simple first step, we envision adapting the usual notion of where-provenance so that each value is tagged with the identity of the user who last set or overrode it. This, for example, would make it easier to see that the unreliable time server in step 7 of the example was due to Alice's change (unnoticed by Bob and Carol), so that Dave should not be blamed.

Dependency tracking [8]. Next, we could consider tagging the final value with a set of all of the users that had some influence over the value (e.g. by specifying or overriding the value). In the example, Carol would see that the time server value was influenced by both her and Alice; this information may make it easier for Carol (or others) to track down the right person to ask about the problem. This type of dependency information is already maintained by the LCFG [1] compiler, but the file-based input allows this to be attributed only at the level of the source file, rather than the user, so it cannot easily be used for access control. It is also susceptible to the previously noted problem where the set of users with some potential involvement in a resource can become very large.

Override history. A refinement of this approach is to replace each value in the system with a sequence of tagged values, with one distinguished, "actual" value and possibly other potential values. The tags indicate which user caused a possible value, and the order of the list indicates the order in which the values were set and overridden. The first element is the current value. In the example, Carol would initially see the list

[`ts@sales.widget.com`{Carol},
`ts@reliable.com`{Alice}],

indicating that she overrode Alice's reliable server. When Alice changes the server to an unreliable one, Carol would see [`ts@sales.widget.com`{Carol}, `ts@unreliable.com`{Alice}], so that Carol may be able to alert Bob of the problem before reverting the sales time server to the default.

In all of the above examples, it is possible that the provenance information could grow much larger than the raw data, leading to information overload. Thus, instead of expecting users to examine all of this information, we envision defining policies that constrain the provenance of the values in the final result, or (extending the approach of Vanbrabant et al.) constrain the changes users are allowed to make in terms of both the data and its provenance.

Either approach, we believe, requires developing a formal semantics for a realistic configuration language, so that we can extend it with provenance and specify desirable security policies or properties. This presents a number of challenges, because (like many languages used in practice) Puppet and similar configuration languages have grown organically, rather than being designed with a formal specification in mind in advance. Defining their semantics formally is perhaps the main challenge for future work.

References

[1] LCFG website, 2012. `http://www.lcfg.org`.

[2] Puppet website, 2012. `http://www.puppetlabs.com`.

[3] ANDERSON, P. *System Configuration*, vol. 14 of *Short Topics in System Administration*. SAGE, 2006.

[4] ANDERSON, P. *LCFG: a Practical Tool for System Configuration*, vol. 17 of *Short Topics in System Administration*. Usenix Association, 2008.

[5] ANDERSON, P. Programming the datacentre - challenges in system configuration. In *Microsoft Technical Report MSR-TR-2008-61 - The Rise and Rise of the Declarative Datacentre* (May 2008).

[6] BUNEMAN, P., CHAPMAN, A. P., AND CHENEY, J. Provenance management in curated databases. In *SIGMOD* (2006).

[7] BUNEMAN, P., KHANNA, S., AND TAN, W. Why and where: A characterization of data provenance. In *ICDT* (2001), no. 1973 in LNCS.

[8] CHENEY, J., AHMED, A., AND ACAR, U. A. Provenance as dependency analysis. *Mathematical Structures in Computer Science 21*, 6 (2011), 1301–1337.

[9] TURNBULL, J. *Pulling Strings with Puppet: Configuration Management Made Easy*. Apress, September 2008.

[10] VANBRABANT, B., PEERAER, J., AND JOOSEN, W. Fine-grained access control for the puppet configuration language. In *Large Installations Systems Administration (LISA) conference edition* (December 2011).