Securing Dataverse with an Adapted Command Design Pattern

Gustavo Durand
Institute for Quantitative Social Science
Harvard University
Cambridge, MA

Michael Bar-Sinai
Computer Science Dept.
Ben-Gurion University of the Negev
Be'er-Sheva, Israel

Mercè Crosas
Institute for Quantitative Social Science
Harvard University
Cambridge, MA

***Abstract*—In order to bake security into application design, we introduce an adaptation to the Command pattern: command instances are tagged with the permissions required to perform them for each object they manipulate. Prior to executing a command instance issued by a given user, an execution engine validates the user has the required permissions over the objects the command is about to operate on. Stating the required permissions can often be declarative. In addition to the usual advantages offered by the command pattern (such as standardized operation handling), this adaptation creates a single checkpoint for validating permissions throughout the application. This, in turn, enhances application security and reduces code duplication, for example between the API and UI controllers. Disadvantages include the lack of framework support, and a learning curve for existing developers. We have used this design in implementing Dataverse, a widely-used institutional data repository developed at Harvard University, which has been in production use since May 2015. As this design differs significantly from common web application design, we also look at how the development team adapted to it, and at how using it affected our development process.**

I. Introduction

It is often argued that application security should be addressed at all stages of design, rather than as an afterthought. While engineering best practices are slowly inching towards this goal, they leave much to be desired. One area where they fall short is enforcing policies at the application-semantic level (as opposed to, e.g., a transport-protocol level). Current practices make it easy to state that *only users with permission P can access resource R*, but more useful statements, such as *to perform action A over objects X and Y, users need permission P1 on X and P2 on Y* are not possible.

This paper proposes a design that addresses this void, using an adaptation of the venerable Command Pattern [3]. Commands, used to encapsulate actions on model objects, are additionally tagged with the permissions required to carry them out, the model objects involved, and the entity submitting them for execution. This allows the permission enforcement logic to be concentrated in a single place in the application, rather than being scattered in many places in code. Thus, the permission enforcement logic is easier to test, and code duplication is reduced. More importantly, the chances of missing a permission check somewhere in the codebase is reduced.

We have been using the design proposed in this paper in production since Mid 2015, in Dataverse, an on-line dataset repository. We can report that we did not find any security vulnerabilities caused by this design, and that the development team was able to adjust to it, even though it is non-standard and has a non-trivial learning curve. This learning curve may be a problem with temporary collaborators, which might be less willing to invest time in learning project-specific patterns.

The rest of this paper is organized as follows: Section II presents the Dataverse system and the security challenges such systems pose. Section III presents the proposed design in detail. Section IV presents our experience using the proposed design in a real-world project. Sections V looks at related work, and Section VI concludes.

II. Meet Dataverse and its Design Challenges

Dataverse [2] [5], developed at Harvard's Institute for Quantitative Social Science (IQSS) since 2006 under co-PIs King and Crosas, is a web-based research data repository. Dataverse enables researchers to share their datasets with the research community through an easy-to-use, customizable web interface, while keeping control of and gaining credit for their data. The underlying infrastructure, whose part of its design is discussed in this paper, provides robust support for good data archival and management practices. The Dataverse software serves as a research data repository in more than 20 institutions worldwide. The Dataverse repository hosted at Harvard University (http://dataverse.harvard.edu) is open to all researchers, and holds more than 340K data files organized in nearly 74K datasets[1].

Design Challenges

Data curation and publishing is a complex domain, swarming with subtleties and inherent complexities. A user can get permissions directly, by being a member of a group, or by using a specific IP address (e.g. when accessing the system from within a university campus or a lab computer). Institutional logins, scripting via API, maintaining auditing log, and journal publication workflows add their own

[1]Figures collected on May 30th, 2017.

complexities. Our deign also had to prepare for supporting multiple data handling policies, towards being DataTags compliant [7], which adds another non-trivial layer of requirements, many of which pertain to access restrictions.

A good data curation system needs to allow accurate specification of permissions users have over data objects, and then to enforce those permissions throughout the application. As these systems offer many features, available both thorough UI and through API, they have a large attack surface that needs to be protected. Having permission checks sprinkled throughout the application creates a security risk, since omitting a single check results in a vulnerability, either for direct data breach, or for permission escalation that can be exploited to create a data breach. The current version of Dataverse contains more than 90K lines of java code[2], and is worked on by core team members and external contributors. Hence, the fear of missing just one check somewhere in the application is not unfounded.

III. Adapting the Command Pattern

The command pattern adaptation we present in this paper circumvents the need for sprinkling multiple permission checks at various place in the code, by directly modeling the notion of actions and the permissions they require. In effect, it allows developers to use Java code to succinctly phrase statements such as *action X on object O can only be issued by users with permission P over O*. This type of formal statements are validated at runtime prior to carrying an action out. In case no required permissions were defined, the system throws a runtime exception. Thus, permissions must be explicitly specified, and are tested in a single place in the code, even though the actions are initiated from multiple places.

In terms of security guaranties, this design ensures that, as long as the described mechanism is used:

1) Users are only able to perform actions they have permissions to perform
2) Developers must explicitly state which permissions are required to run each command
3) To the extent that static permissions declaration is used, the permissions required to run each command are documented in the code.

The described design protects against security issues caused by accidental developer oversight or negligence. It does not protect against malicious code or interference with underlying data stores (e.g. direct database access).

The classic Command pattern, as defined by Gamma et. al. in [3], "encapsulates a request as an object". In the context of an application like Dataverse, a *request* means "an operation on model objects", such as a dataset. It involves *receivers*, the model objects which are acted upon, *concrete commands*, which implement operations on the receivers and implement the *command* interface, an *invoker*, which invokes the command, and finally the *client*, which is the code that creates the commands and submits them to the invoker.

In the context of Dataverse, the receivers are the model objects: Dataverse, Dataset, and DataFile. For familiarity reasons the invoker is called "Engine". Our code contains two implementations of such engine: one for unit tests, and one for normal application use. Finally, there are three types of client code that invoke commands: the back-end of the user interface ("backing beans" in Java EE terms), the API implementation code ("JAX-RS beans"), and system events that run on a scheduled timer.

The classic pattern already achieves a few of our design goals:

1) **Code reuse across UI and API** As operations on the models are performed by command instances, all client code has to do is to generate appropriate command for the user request, and submit it for execution. As interpreting user request in the UI is very different from interpreting it in the API, there is no code duplication in the interpretation part.
2) **Logging** Commands work at the application-semantic level (as opposed to HTTP and database logs, which can only serve as a proxy to it). Thus, a command log is a strong tool for auditing the activity history of a repository. When using the Command Pattern, commands are executed at a single location in the code, and have a common set of fields. Thus, logging all commands — in effect, logging all operation on the model objects — becomes an almost trivial task.

However, the classic Command Pattern leaves a few things to be desired. First, a way of enforcing permissions: the design needs to ensure that users can only perform certain operations on certain model objects — exactly what operations and which objects should be decided at runtime, based on the user's permissions on a given object. Second, as the commands run in an application container, the design needs to make the resources supplied by the container available to the commands — if it decides to invoke them.

The second goal is easily achievable by introducing a `CommandContext` parameter to the command's `execute` method. A context object allows commands to interact with server resources (mainly database operations) via its methods. This design also lends itself to unit testing: all a test engine has to do to test a command is to supply it with a context object that links to mock resources. For achieving the first goal, though, we need to add some metadata to the concrete commands.

A. Annotating Required Permissions

The three pieces of information needed to decide whether a user can perform an action — which translates to executing a command instance on model objects — are a) which model objects are involved, b) what permissions is the user required to have over each of these objects, and c) what permissions does the user actually have over them.

[2]Statistic generated using David A. Wheeler's 'SLOCCount'.

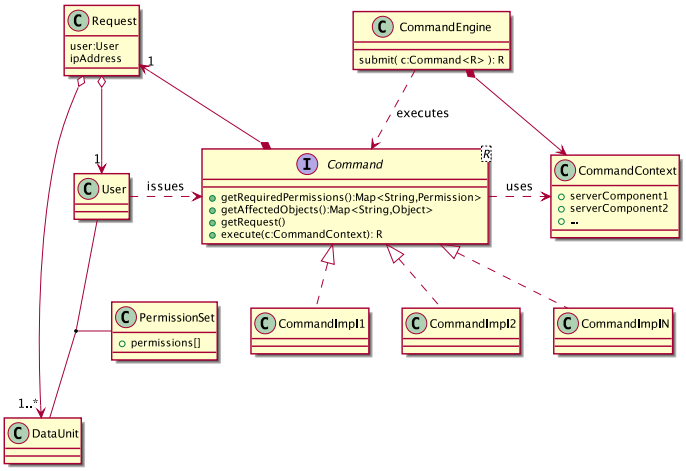

Fig. 1. Participants of the Secure Command pattern. In addition to the added security methods, the interfaces of the classic Command have been modernized a bit. The `execute()` method returns a value, so the `set(command); execute(); result = get()` sequence in the original command, which is based on state, can be replaced with `result=execute(command)`. Some classes were removed for brevity.

Given this information, the command engine can decide whether to execute or reject the command. Figure 1 shows a class diagram of this design.

To add metadata to the commands, we have added three methods to the "classic" command object:

1) `getRequest` Returns the request context of the command, which includes the user and some request metadata, such as a source IP address.
2) `getAffectedObjects` returns the objects the command would operate on.
3) `getRequiredPermissions` returns the permissions required to operate on the affected objects.

The two latter methods return maps, mapping a key with semantic significance to their respective values (model objects and sets of permissions). For example, a `copy` command may require a `READ` permission on the `''source''` object, and a `WRITE` permission on the `''target''` one. Thus, the `requiredPermissions` map would include the bindings $source \rightarrow READ, target \rightarrow WRITE$, and the `affectedObjects` map the bindings $source \rightarrow$ `<source object>`, $target \rightarrow$ `<target object>`.

For a given command, stating which permissions are required over what target objects should ideally be done in a declarative way. This makes it easier for developers to assign the correct permissions, and for code reviewers and re-users to see what permissions are needed at a glance.

To this end, we have developed a set of class-level annotations[3], that allow developers to statically assign required permissions to specific command subclasses. For example, when annotating a concrete command class with a `@RequiredPermissions(Permission.PublishDataset)`, the developer states that instances of this command will be executed only for users who have a *PublishDataset* permission over the affected objects. Listing 1 shows declaration and usage of a typical command class that uses statically specified permissions[4] .

Listing 1. Declaration and usage of a concrete command class that uses statically specified permissions. The **`GetDraftDatasetVersionCommand`** class is a command class that retrieves an unpublished version of a given dataset. As these versions are not public, they can only be viewed by users who have the `ViewUnpublishedDataset` permission over the dataset whose draft version is being retrieved. This is declared as a class-level annotation (line 2), and enforced by the command execution engine (line 10).

```
// class declaration                                                    1
@RequiredPermissions( Permission.ViewUnpublishedDataset )               2
public class GetDraftDatasetVersionCommand                              3
                  extends AbstractCommand<DatasetVersion>{              4
  ...                                                                   5
}                                                                       6
                                                                        7
// class usage                                                          8
try {                                                                   9
    Dataset d = engine.submit(                                          10
            new GetDraftDatasetVersionCommand(user, ds));               11
} catch(CommandException ex) {                                          12
  ...                                                                   13
}                                                                       14
```

Commands that work on multiple model objects require a more complex annotation, as the user might need different permissions over each object to be allowed to execute such command. This is achieved by specifying a map whose keys are model object role names, and values are required permissions. Listing 2 shows an example of static permission specification for more than a single object, using the `@RequiredPermissionsMap` annotation. In effect, `@RequiredPermission(X)` is a shorthand for `@RequiredPermissionsMap{@RequiredPermission(name="",value=X)}`.

Listing 2. A concrete command class that uses complex statically specified permissions. The class annotations specify the map of object role names to required permissions (lines 1-8). The class constructor maps the same object role names to actual model object instances, using a map generator mechanism called `dv` (lines 13-15).

```
@RequiredPermissionsMap({                                               1
  @RequiredPermissions(name="moved",                                    2
                       value=Permission.GrantPermissions),              3
  @RequiredPermissions(name="source",                                   4
                       value=Permission.UndoableEdit),                  5
  @RequiredPermissions(name="destination",                              6
                       value=Permission.DestructiveEdi )                7
})                                                                      8
public class MoveDataverseCommand extends AbstractVoidCommand{         9
  public MoveDataverseCommand(User aUser,                               10
                              Dataverse moved,                          11
                              Dataverse destination ) {                 12
    super(aUser, dv("moved",       moved),                            13
                 dv("source",      moved.getParent()),                14
                 dv("destination", destination));                     15
    ...                                                                 16
  }                                                                     17
```

Stating permissions declaratively has its downsides, though. For example, it may lead to code duplication when required permissions depend on application state. Consider a command that retrieves a dataset. Before executing the command, the command engine has to validate that the requesting user has the required permissions to view said

[3]In Java, class annotations could be thought of as machine readable code comments that can be available at runtime.

[4]Code listings in this paper were slightly altered for brevity and clarity. Actual code is available at https://github.com/IQSS/dataverse/.

dataset. These permissions depend on that dataset's state. Specifically, on whether that dataset is published, or still being worked on.

It is possible to accommodate this situation using declarative, class-level annotations only: One can create an abstract base class for retrieving a dataset, and create two subclass, one for published datasets, and another for unpublished datasets. The dataset retrieval code would reside in the base class, and the subclasses will only contain the permission annotations. However, this approach results in three classes instead of one, and requires correct selection of the concrete class at every place in the application where a dataset is retrieved. Furthermore, in the general case, where combining multiple permissions, this may lead to an exponential number of subclasses.

Thus, our design allows command objects to specify their required permissions dynamically, when needed. This is done in the normal object oriented manner: The abstract command base class builds a required permission map based on the annotations of the instance's concrete class. Concrete commands may override the relevant methods and decide on the required permission map based application state at runtime (see Listing 3 for an example). Our experience shows that while dynamic permission requirements are useful, most cases can be accommodated using the static, class-level declarations (see Table IV-B).

Listing 3. A command may specify the its required permission dynamically. Shown here is the permission specification logic for `GetDatasetCommand`, a command that retrieves a dataset. The application requirements are that a published dataset may be viewed by anyone, whereas viewing non-published datasets requires a special permission. Thus, for unpublished datasets, this logic returns a set containing the required permission. For published datasets, the method returns an empty set. Note that it is not acceptable to return `null` in this case, since the engine would interpret this as a programmer error of not specifying required permissions.

```
1  public class GetDatasetCommand implements Command<Dataset>{
2    ...
3    @Override
4    public Map<String, Set<Permission>>
5                                  getRequiredPermissions(){
6      if ( dataset.isReleased() ) {
7        return Collections.<Permission>emptySet();
8      } else {
9        return Collections.singleton(
10                       Permission.ViewUnpublishedDataset));
11     }
12   }
13 }
```

B. Command Execution

When client code submits a command to the engine, the engine first reads the command's metadata, and tests whether the request meets the requirement posed by the command: does the user have the correct permissions over the affected objects? If the request meets the requirements, the engine invokes the command. Otherwise, the engine throws and exception stating which permissions are missing over what receiver objects (see Figure 2).

This design can easily be extended to support more complex data handling policies. Consider a new requirement, stating that some datasets are can only be viewed through a secure connection. To implement this, one needs to add

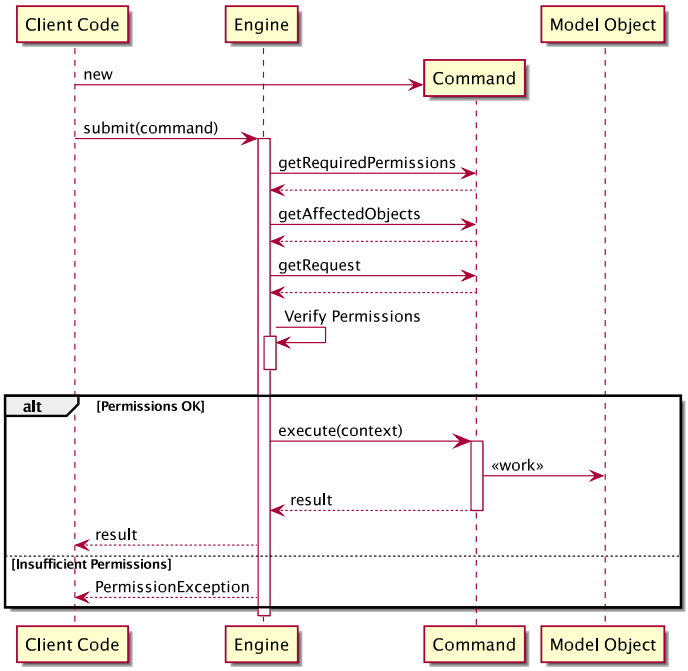

Fig. 2. Sequence of executing a command. The client code creates a command and submits it for execution. The Engine verifies that the user has sufficient permissions to issue the command over the model objects. If so, the engine executes the command and returns the result. Otherwise, an exception is thrown, detailing which permission is missing.

an `isSecureConnection` flag to the request, and update the engine's logic. The rest of the code remains unaffected.

C. Client Code Examples

All permission verification is done at the command execution engine. Thus, client code using the engine is free to focus on creation and submission of commands, rather than on permission checking and model updates. Thanks to the modernized command interface API (see Subsection III-A) and to the usage of Java's generics, it is easier for developers to use the framework than it is to bypass it. This is important, since after learning to work with the updated command pattern, the path of least resistance for developers is also the correct one. This section contains examples of client code using a command engine.

Simple Command Submission

```
try {                                                          1
  engine.submit(new DeleteDatasetCommand(user, dataset));      2
} catch (CommandException ce) {                                3
  ...                                                          4
}                                                              5
```

Deleting a dataset is an example of a basic command usage. The client code retrieves the dataset object and the user, generates a new `DeleteDatasetCommand` instance with using both, and submits it to the engine for execution. The engine will check whether `user` has the correct permissions over `dataset`, and execute or discard the command instance accordingly. Discarding the command will result in an

exception of class `PermissionException` being thrown. This exception is a subclass of `CommandException`, and thus will be caught by the **`catch`** block, giving the client code a way of reporting the problem back to the user.

Retrieving a List of Model Objects

```
1  engine.submit(new ListVersionsCommand(user, dataset))
2        .stream()
3        .map( d -> json(d) )
4        .collect(toJsonArray()))
```

The above code generates a JSON[5] string containing the list of versions of a dataset. Since the static return type of `ListVersionsCommand`'s `execute` resolves to a list of dataset versions (`List<DatasetVersion>` in Java notation), the entire process of command generation, submission, permission validation, and execution can be performed in a single Java expression. The result of this expression may be passed to a containing expression or a calling function.

Higher Order Commands

```
1  @RequiredPermissions({})
2  public class GetLatestAccessibleDatasetVersionCommand
3                   extends AbstractCommand<DatasetVersion>{
4    ...
5    @Override
6    public DatasetVersion execute(CommandContext context)
7                                   throws CommandException{
8      DatasetVersion result = null;
9      try {
10       result = user, dataset.engine().submit(
11          new GetDraftDatasetVersionCommand(user, dataset));
12     } catch(PermissionException ex) {}
13
14     if (result == null) {
15       result = dataset.engine().submit(new
16        GetLatestPublishedDatasetVersionCommand(user,
17                                                dataset));
18     }
19     return result;
20   }
21 }
```

Commands can be composed of other commands. This allows developers to re-use the permission verification logic as part of regular business logic. The above example, shows a command that retrieves the latest dataset version a user is allowed to view. This is implemented by first attempting to issue a command that retrieves an unpublished version of the said dataset (lines 10-11). This operation may fail for two reasons: a) there is no such version, or b) the user is not allowed to view it. Either way, if no dataset version was retrieved, the composed command retrieves the latest published version (lines 15-17).

The above command does not require any permissions to run, since no permissions are required to view published datasets. Thus, it is annotated with an empty `@RequiredPermissions` (line 1). Omitting this annotation would result in a runtime error, where the command engine complains that no required permissions were specified for the command.

[5]A textual data interchange format, acronym for JavaScript Object Notation. See http://json.org.

IV. The Pattern in Practice

The largest effort of course, was coding the engine and infrastructure for the pattern. When it was finished, we provided it and a few basic examples to the developers.

We started by taking existing code and refactoring it to use the new Pattern. This was overall straightforward, as a new basic command is fairly simple. The guts are really in the business logic, and for existing code, we were able to transfer much of that from the classic business logic layer[6] to the Command classes.

The team was already well experienced with annotations in general, so explaining how to assign the correct permissions via annotation was also relatively straightforward.

One challenging part to teach was to provide a good understanding of each of the different kinds of exceptions, and when it was proper to use each one.

Another minor challenge was with making sure developers would use this new Pattern with new functionality. It was not so much an issue of how to use it, but in making sure to provide the motivation, as the effort for using the new Commands was (marginally) more than the traditional effort. Once it was clear, though that the new Pattern provided a solid architecture and actually made code maintenance easier (for example, if a permission needed to be modified), the developers were generally enthusiastic.

Using the adapted command pattern became more of an issue with external development. In one collaboration, code was written for an alternative way to edit datasets that did not use the pattern. It worked well, but it was redundant in a lot of permission checking. The problem with this was that if we decided to change the required permission for dataset editing, we would need to remember to change more than just the Command.

In this particular case, the external developer was hesitant to change the code to the command pattern, as it worked well for what they needed and there were many other requirements to be completed. Since merging their changes was critical part of the project we were able to convince them to make the change. Unfortunately in doing that a new performance issue arose, and rather than attempt to solve it by fixing this issue with the command, the developer chose to revert to the previous code.

This is likely to be a rare case, however, as we later learned that the developer was leaving the project and wanted to reach a certain milestone before his final date. To be sure, though, with future external development we need to make sure we introduce the reasoning and benefits of the new Pattern upfront, so that the functionality gets originally coded this way and does not have to be rewritten.

Additionally, in the beginning we wrote too many new commands and/or duplicated the same functionality in multiple commands, instead of trying to find commonality

[6]As Dataverse is a Java EE application, this layer was implemented using Enterprise Java Beans (EJBs).

between existing commands. One example of this is that we have separate commands with duplicate code for Create Dataset, Update Dataset, Create Dataset Version, and Update Dataset Version. One code cleanup task we have is to chain these commands to call each other, for example have Create Dataset calls Create Dataset Version calls Update Dataset Version, which reduce a lot of the duplicate code.

While developing these commands in practice, this is how we discovered that in addition to the static permission annotation, we needed the ability to define the permission at runtime. We solved this with the dynamic permissions previously described.

From a performance standpoint, the presented design does not incur a substantial overhead for most operations, as the permission testing has to be performed whether this design is used or not. One case where manually written permission checks can significantly outperform the command pattern is when the same user's permissions are being repeatedly tested against the same model object, for example when running a command in a loop. In such cases, manually written code is likely to calculate the permissions only once, before the loop starts.

One way of tackling this issue is by adding short-term permission cache to the engine.

Overall, the new Command pattern has been a good addition to our core infrastructure. If we spend some time up front onboarding new developers and external collaborators, the effort to use the new pattern is minimal, while the benefits are substantial.

A. Security Vulnerabilities And the Command Pattern

At the moment, Dataverse is not meant to store sensitive data. As such, Dataverse instances are not subject to detailed security scrutiny. However, as an open-source project with more than 20 installations worldwide, it enjoys an involved community that reports bugs and vulnerabilities, in addition to security issues found by the Dataverse development team. Over the years, some vulnerabilities were found. Examples include tokens leaked by a system interfacing with Dataverse, cross site scripting (XSS), issues stemming from included libraries (such as Weld) and from using the Glassfish server, and a vulnerability that allowed an attacker to pretend a request is coming from a specific IP address, when a Dataverse system was behind a proxy server. This was possible by forging the `X-FORWARDED-FOR` HTTP header. The latter was found by a community member who reported an issue through the projects's repository[7].

However, none of the issues reported so far was caused by the command pattern adaptation presented here.

[7]https://github.com/IQSS/dataverse/issues/2826

B. Command Statistics

At the time of this writing, the development version[8] of Dataverse contains 73 classes implementing the `Command` interface. Table IV-B lists some statistics about their traits.

TABLE I
STATISTICS ON COMMAND CLASSES IN THE DEVELOPMENT VERSION OF DATAVERSE, MAY 28, 2017.

Command interface implementations	73
Abstract	3
Concrete	70
Uses dynamic permissions	8

V. RELATED WORK

Enforcing permission-based policies is an issue both industry and academia have been dealing with for a while. On the industrial side, modern application authentication and authorization frameworks such as the one offerred by Java EE[9] or Deadbolt2[10] for Play framework[11] have solved authentication and role-based permission assignment. These are the basic building blocks that allow application developers to verify a given user was granted a certain permission. However, this check has to be performed at each location in code where an action that required said permission is attempted. Furthermore, stating for permissions in code is harder to review and audit, compared to declarative statements.

Protecting web resources (such as URLs) can be done with declarative syntax (for example, using XML in Java EE and method composition in Play). These approaches are well suited to protect on-line resources (e.g. web pages) from unauthenticated users, but they do not operate in the application-semantic level, and the roles are system-wide rather than in the context of specific model objects. In other words, they are good for stating *Only users with role X can access resource Y*, but not for stating *Users need permission Y over object Z in order to perform action W*, which is what the proposed design allows developers to state.

Academic work offers detection of data leaks using formal methods. PIDGIN [4], by Johnson et. al., analyses data flow in an application, thus allowing developers to detect leaks and enforce access policies. Yang et. al. [8] use dynamic information flow control to formally guarantee program properties such as non-interference. In [9], Yang et. al. present a functional constraint language named *Jeeves*, which allows tagging variables as sensitive, and conditionally redacting them. These approaches are more oriented towards data flows within an application, and less towards actions on model objects.

[8]Dataverse is developed using Git with a per-feature branching strategy. This, this term refers to the *develop* branch in the main Dataverse repository.

[9]http://www.oracle.com/technetwork/java/javaee/overview/index.html

[10]https://github.com/schaloner/deadbolt-2

[11]https://playframework.com

Moore et. al. [6] present the concept of *Authority Environments* which uses execution contexts and associates them with rights. This allows extensible access control using software contracts, and can also restrict actions.

Another project that adds security guarantees to an application is RESIN [10]. RESIN takes a complementary approach to the command pattern adaptation presented here, by focusing on data flows rather than on actions. While giving stronger security guarantees, RESIN is a language runtime rather than a design pattern. Each in its own way, both projects allow developers to declare security-related policies. Passe [1], which also aims to restrict illegal data flows within an applications, takes a different approach on specifying a security policy: it uses learning rather than explicit specification by a developer.

VI. Conclusion

In this paper, we present an application design that enforces permission-based policies, based on an adaptation of the classic Command Pattern. By allowing commands to specify a) which permissions are required to execute them, b) which model objects are involved, and c) what is the context in which the command was submitted, a central command execution engine can verify that a command should indeed be executed prior to invoking it. We have used this design in practice since 2015 in Dataverse, a web-based research data repository platform. Refactoring Dataverse to use this design was not trivial, both in terms of coding and in adapting the development team to use it. However, almost three years in, this architectural switch has proved itself beneficial.

Acknowledgment

The authors thank Stephen Chong for reviewing an early version of the proposed design, our anonymous reviewers, and the entire Dataverse team at IQSS.

References

[1] Aaron Blankstein and Michael J Freedman. Automating isolation and least privilege in web services. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 133–148. IEEE, 2014.

[2] Mercè Crosas. The dataverse network®: an open-source application for sharing, discovering and preserving data. *D-lib Magazine*, 17(2), 2011.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[4] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *ACM SIGPLAN Notices*, volume 50, pages 291–302. ACM, 2015.

[5] Gary King. An introduction to the dataverse network as an infrastructure for data sharing. *Sociological Methods and Research*, 36:173–199, 2007.

[6] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible access control with authorization contracts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 214–233. ACM, 2016.

[7] Latanya Sweeney, Mercè Crosas, and Michael Bar-Sinai. Sharing sensitive data with confidence: The datatags system. http://techscience.org/a/2015101601/, 2015.

[8] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. End-to-end policy-agnostic security for database-backed applications. *CoRR*, abs/1507.03513, 2015.

[9] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.

[10] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009.