Role-based Access Control and BCHS

Kristaps Dzonsons
The BSD.lv Project

Abstract

Web applications present an attractive attack surface in part since they are public front-ends to valuable data sources. Not only are these applications network-facing, they must also accept a non-trivial set of inputs, perform complex tasks, and produce diverse outputs—each of which may be manipulated by a skilled attacker. While a great deal of active research concerns itself with restricting system resources from attackers, there are few resources for protecting an application's *internal* data sources. In this paper, I describe how recent developments in BCHS web applications allow programmers to define, enforce, and audit access roles of the application and its data source. These developments bring hard guarantees on data security from the application scope to the operational scope.

1 Introduction

Consider a web application *foo* that provides a marketplace for buying and selling widgets. Registered clients log in to *foo*, post widgets for sale, browse markets, and acquire new widgets. Unregistered users can browse as well, albeit with less information on the widgets and principles. Administrators also play an important role in curating the marketplace and performing routine maintenance.

In the interests of simplicity, let's assume payments, warehouse management, and other such tasks are handled elsewhere.

To render its services, *foo* interacts with a database defining its registered clients and public users, sessions, administrators, inventory, and other market information—spot prices, fees, etc.

The *foo* development team clearly has a formidable task: not only must the "business logic" of the marketplace be well-defined and properly written, but there must be clear separation within the marketplace of user classes to prevent collusion and other adversarial behaviour that affects the market economy.

They must also guarantee that the command parameters that govern this business logic, such as identities and operations, must be properly derived from the diverse and inter-connected set of input modalities: CGI, HTTP, JSON, etc. Any of these may be adversarially altered, separately or in concert, to craft a false operational environment.

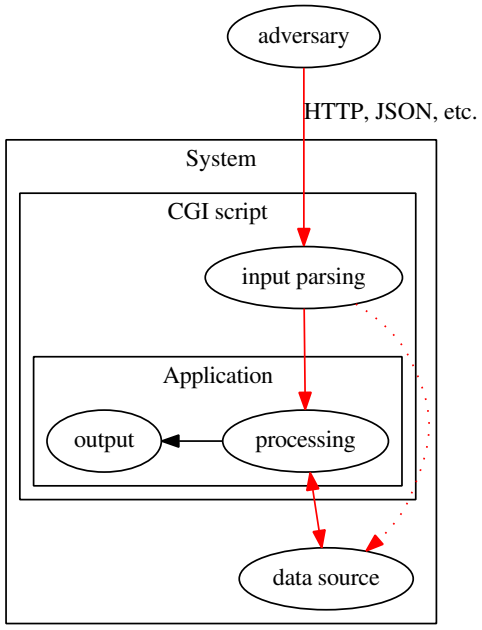

Figure 1: Layer separation and attack surface.

In Figure 1, we see the OSI model's application and presentation layer as input parsing, then processing and output, respectively. A clear line of attack flows from the adversary indirectly to the data source, and also directly from compromised input sources.

Given the promulgation of similar marketplaces, it's

reasonable to acquire one of the many COTS solutions, configure to the services rendered, deploy, and hope for the best.

However, consider the service that *foo* renders: an online gold exchange.

The inventory managed by the database is that of warehoused metals, and the buying and selling process deal with high-cost exchanges between principles whose privacy is also a premium asset.

We often described security as rendered in the context of generic applications exchanging widgets, but many of the applications we depend upon on a daily basis broker similarly high-value exchanges: personal and business banking, stocks and commodities, bill-paying, sensitive communications, and so on.

When considering *foo* as a gold market instead of a widget market, or substituting "widget" for any high-value artefact, we can appreciate the development team's necessary focus on security. With a single troy ounce of gold being over 1 000 USD, it's easy to see that *foo* will have its attack surface carefully examined by skilled adversaries.

Fortunately, application security is a well-researched field—on all levels of the OSI stack.

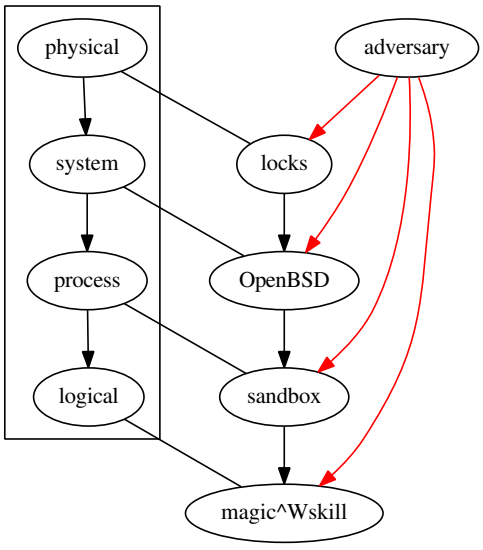

Figure 2: Counter-measures for adversarial behaviour.

When considering a security process for the needs of a gold market, we must carefully construct from the ground up with counter-measures on each level of granularity. Physical security (the "media layer", in OSI parlance) is beyond the scope of this paper, as is general operating system security (layers 4–6). Securing an application from its operating environment is well-studied, especially in recent years with sandboxing technologies, privilege separation, ASLR, and so on.[5][4]

In this paper, I introduce a method and implementation for provisioning logical (*internal*) access to a web application's data source with role-based access controls, or "RBAC". This method will extend our security measures from the process scope—with all of its robust measures of ASLR, sandboxing, etc.—to that of its "business logic".

What is role-based access control? A *role* in a RBAC system is an ontological object representing the operator of an application. For example, a web application process typically begins in an unspecified role. The user is then authenticated in some fashion—say, a database query on a session object or login credentials—with a role derived from the authentication. This might be that of an administrator, or a registered user with buyer capabilities, or an anonymous browser.

Roles are well-suited to web applications since they often classify users as-is. Sometimes this is implicit in the database structure, sometimes explicit in an enumeration of user types. This allows users to join and leave the system without needing to account for individual access rights—we negotiate access in terms of the user's class, not their identity.

Moreover, additional features, such as adding an administrative role for warehouse management, require only adding to the enumerations of possible roles. Or in the implicit case, adding additional tables describing the role and linking them to an abstract user table.

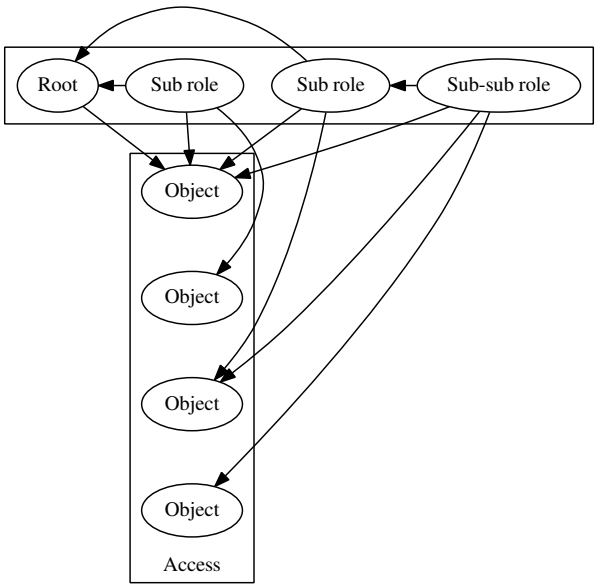

Figure 3: Hierarchical roles and objects.

Once roles have been defined, *access* must also be defined. In this paper, we discuss access in terms of

database operations—SQL statements—and the propagation of exported data. To wit, we'll pursue role assignment to database queries, database modifications, and export of data fields.

By unambiguously mapping our application's user roles into database statement, we'll exploit existing privilege separation strategies to *control* database access by access role.

Our implementation is not the first of its class, but presents a simple and novel way of establishing guarantees on the operation of given roles using conventional methods.

2 History

The concept of formal access control has a long history. The initial public standards were derived in the late 1980s from prior work in the late 60s and 70s, most of which being related to the access of classified materials for government and military use.[9] The access methodologies of this age were limited to *discretionary* and *mandatory* access controls,[7] culminating in the "Orange book" (TCSEC, 1983, updated in 1985).

Discretionary controls match our understanding of the popular `doas` and `sudo` applications, which grant access to objects (including operations) at the discretion of an out-of-band process. UNIX permissions are themselves DAC systems, since permissions are set (at the discretion of) the user.

Figure 4: Check at discretion of object's owner.

The concept of mandatory controls allows an object (commonly as found in the file-system, but generalisable to any object) to itself define the level of access. Of the BSD systems, mandatory controls are only available on FreeBSD by way of the POSIX.1e extensions via the TrustedBSD Project.[14][13]

Figure 5: Check against mandatory policy.

Discretionary access control is well-suited to "drive-by" security requirements, such as was initially implemented between security-concerned organisations and temporarily-attached bodies such as contractors and consultants. It also fits well into collaborative environments, such as on multi-user academic systems, where sharing of documents is at the discretion of the object owner.

In the government and military systems described in the literature, DAC systems were often augmented with MAC to enable multi-level classification protection. This made certain resources have distribution fixed by mandatory policy, while others were at the discretion of the owners.

Similar environments may be created in today's FreeBSD by making use of its POSIX.1e extensions for mandatory access control alongside the conventional user permissions.

The foundational models stimulated a significant number of additional security models and implementations.[2] Their mention here is in their importance in laying the foundation for access methodologies more germane to the needs of the *foo* web application.

It's clear that neither of the given methodologies succinctly meet the needs of the *foo* web application. It is neither appropriate for the principles of the system (administrators, registered clients) to govern discretionary access to data, nor for a mandatory policy regarding individual principles.

In 1992, Dr Kuhn formalised the concept of *role-based* access control, which expands the concept of mandatory controls to accommodate for roles of principles, instead of the principles themselves. This is a useful generalisation for individuals who usually are already organisationally classified within departments or sub-institutions. Moreover, RBAC can gracefully degrade into MAC by having each user within its own role.[6][12]

As in the prior MAC formulation, RBAC is also non-discretionary.

Moreover, by explicitly accommodating for a concept of available and active roles, RBAC allows for all manner of inheritance topologies. For example, it's simple to enact a framework of hierarchical roles, where lesser roles (e.g., an administrator for users being a lesser role than an administrator) drop the privileges of the parent. It's similarly possible to enact the opposite, where lesser roles gain certain privileges, such as the same administrator for users being able to access certain database tables not accessible from the general administrator class.

RBAC was rigorously defined as an ANSI/INCITS standard (NIST RBAC) in 2000[11] and further in 2004, and has since then seen wide-spread adoption in many environments. In the coming section, we investigate whether RBAC is available in systems we may use for deploying the *foo* web application.

3 Related Work

In considering related work, we restrict our survey to modern, well-known open source systems that may be applied to the *foo* web application as described. This limits us to databases, frameworks, programming languages, and programming language libraries (third-party components, depending on the parlance of the language).

A limited concept of roles is specified in SQL:1999[8], with varying levels of implementation across popular systems. Both PostgreSQL[1] and MariaDB[2] implement a form of role-based access control. SQLite does not have this native functionality[3].

We refer to PostgreSQL version 10.1, MariaDB version 10.2, and SQLite version 3.22.0 in this paper.

MariaDB has a simpler implementation, as it does not allow for role privilege inheritance. However, it's in theory possible for a third-party application to manually construct hierarchical MariaDB roles from a third-party specification.

More significantly, MariaDB does not support reentrant role setting: roles are mapped directly to user classes; and while a user's role may not be changed, the current role itself may not be granted a further-restricted set of available roles. We consider this a serious limitation; as in practise, a system generally begins with broad permissions, then narrows them as it completes its task. (Or the opposite.)

However, the MariaDB implementation does allow for restricting roles to a set of stored procedures, which in theory can be used to describe all possible fine-grained operations available to a role, the PostgreSQL implementation only restricts to classes of SQL statement. Both allow for granularity on the level of table columns.

While we consider the concept of hierarchical roles to be significant, any system with hierarchical access rights may have equivalent semantics. In such a situation, a role instead is granted with significant privileges, then drops privileges as its role narrows (having started with the inability to grant further privileges). However, this lacks the simplicity of guaranteeing unequivocally that a process in a given role has known privileges.

In the field of non-RDBMS systems, MongoDB[4] has support for RBAC on par with PostgreSQL and MariaDB. The support for RBAC in other so-called "NoSQL" systems, which lack standards-based syntax, are diverse and beyond the scope of this paper. Like MariaDB, however, MongoDB roles are assigned to a given user and transition between roles is not well-defined. Privileges may, however, be inherited and dropped during run-time.

The current version of MongoDB referenced in this paper is 3.6.

It's clear that RBAC has an extensive history of implementation in standards-based SQL databases, with considerable implementations prior to that as well[3]. However, one of our motivations is to make the concept of roles extend into our application logic, allowing us to control both for the access of data as well as establish limitations on its export. Thus, we turn to RBAC implementations available directly to the "business logic" of our web application.

The usage of RBAC for fine-grained control within running applications also has known implementations[1]. In surveying availability, we limit ourselves to modern open source systems. In surveying these fields, it's clear that the following is true:

> However, and despite the few methods that attempt to automatically derive these policy implementations from high-level security specifications, the task of implementing an access control security policy remains in the vast majority of cases a manual process which is time-consuming and error-prone.[10]

The PHP language has external support for roles [5], as does Python[6], Golang[7], and Perl[8]. All of these are popular languages for web application development, and the security of the database is often cited as one of the driving reasons for RBAC implementations.

Many other popular languages have similar levels of support, and there are many other implementations for the above languages—far too many to mention in this paper.

Regarding the usage of roles and observing the proliferation of role concepts in SQL:1999 implementations, it's also conceivable that a system could be written that extracts role information from a role-enabled database (such as PostgreSQL) and feeds that into the PHP, Python, or Perl run-time role system. This would provide a complete solution.

A significant issue with RBAC implementations in these languages, and indeed in *any* language or system where the RBAC is enabled within the process itself, is that a sufficiently-motivated adversary might be able to disable these features from within the application as well by manipulating the process space. In other words, there

[1] https://postgresql.org

[2] https://mariadb.org

[3] "Since SQLite reads and writes an ordinary disk file, the only access permissions that can be applied are the normal file access permissions of the underlying operating system. The GRANT and REVOKE commands commonly found on client/server RDBMSes are not implemented because they would be meaningless for an embedded database engine." (https://www.sqlite.org/omitted.html)

[4] https://www.mongodb.com/

[5] http://phprbac.net

[6] https://yosaiproject.github.io/yosai

[7] https://github.com/mikespook/gorbac

[8] http://perldancer.org

is no guarantee of the system's operation beyond those of the language designers.

We consider this a significant concern due to our desire to make guarantees on the availability of data to given roles. The database methods, as described, avoid this failing by having the database managers run in a process apart from the web application. (SQLite, which is designed for in-process use, is an exception but has no native RBAC support to speak of.)

While it's certainly possible to implement a compartmentalised security process for any of the above languages, none exist to date. There is clearly some room for improvement: being able to guarantee that a sufficiently-talented adversary cannot compromise our process and affect our roles, and having the simplicity of reentrant roles with reduced privilege semantics.

4 Method

I now describe a method and implementation of assigning, enforcing, and auditing role-based access control. I use as my exemplar utility our gold exchange web application, *foo*, to emphasise the demands on security.

User demand on the system is not significant (bandwidth, concurrent users, storage requirements, etc.), but as we wish to use conventional hardware (VMs), system resource usage should be conservative. In other words, the application will be in the demand class of most small to medium-sized web applications, on the order of thousands to tens of thousands of total users having negligible concurrency demands.

The specification starts with the stipulation that we must work with a stock BSD system with minimal added configuration. We choose OpenBSD due to familiarity—FreeBSD has similar security features for our needs. (NetBSD, however, does not.) This paper does not cover the merits of various operating systems on the system level of security, nor does it expand upon the many security measures enacted around processes.

While we acknowledge the prior existence of RBAC utilities in the PHP, Perl, and Python languages, we use C as our basis languages. The choice of C will become more apparent when we consider the implementation details of establishing guarantees; but in a more general sense, the simplicity of RBAC allows us not to depend on prior implementations. With the assumption that any programming language is equally secure in the hands of a skilled developer, we allow our constraints of few dependencies and deployment simplicity to guide our choice.

Further, though PostgreSQL is an attractive choice due to its native RBAC functions, we use SQLite as a database engine. The reason for this follows our choice of C; however, we also acknowledge that future data requirements may stipulate a change to PostgreSQL, and thus make our choice of database to be agnostic in that it will be hidden behind an access layer.

Our choice in SQLite bears further mention since PostgreSQL has some clear advantages in prior support of RBAC. While using RBAC on a table access level is attractive, this feature is also available from SQLite (as subsequently discussed). Moreover, the general use of PostgreSQL roles is on a modification level (whether `insert`, `update`, and so on), while our access controls operate on a per-statement basis. MariaDB has support for storing statements that may be access-controlled by roles, but also lacks reentrant roles.

In short, the choice of PostgreSQL and MariaDB both require external scaffolding to augment roles: it stands to reason that if we're to have a scaffolding, the initial database should be as simple as possible.

As mentioned earlier, SQLite does support access authorisation with the `sqlite3_set_authorizer` function. Our implementation does not currently use this feature, as access is limited to SQL statements and not specific data, but anticipates doing so as described in the Future Work section.

The last component required for *foo* is the ability to audit roles. This is critically necessary to being able to maintain an access policy across releases and development teams. In the event of external audits, a meaningful illustration is important to begin considering the availability of data.

None of the available mechanisms, neither as a programming language library nor within the database, have this possibility. This further reduces the unique advantages of the existing implementations, since auditing is a non-trivial human-interface (or automated) task.

The responsibilities left to the *foo* development team, with these methods well-known, are to implement the solution.

5 Implementation

We begin by using kwebapp[9] to implement our data layer. kwebapp generates the API and implementation of a data model specification, input and output, allowing us to focus on the "business logic" of our market transactions. It also allows us to specify roles entirely within a configuration file—and audit the access of those roles.

In using this system, our development team need only to define the data model, roles, then create routing logic mapping command parameters to defined functions.

The backend of kwebapp is pre-defined as ksql[10], which is currently a shim for SQLite. Our primary security measures will take place during the interaction be-

[9] https://kristaps.bsd.lv/kwebapp
[10] https://kristaps.bsd.lv/ksql

tween kwebapp and ksql. It's worth noting that, while ksql is concerned primarily with SQLite, it's not by design limited to any database. We leave interacting with a PostgreSQL or MariaDB database as an inevitable exercise for future expansion.

The security of *foo* will be enforced largely by use of OpenBSD's pledge[11] and privilege separation[5].

After parsing command parameters using kcgi[12], kwebapp uses ksql to open its database in split-process mode. The split-process mode of ksql spawns a child, which it immediately pledges to only access the database, then waits for instructions by which to manipulate the database.

Once the database has been opened, the calling application, which embodies the business logic of *foo*, pledges itself to have nothing but access to the socket communication of output (JSON or HTML) and the database. This prevents compromise to our application logic from affecting the database directly—it significantly protects our database from internal corruption.

Once the database is open, we know the following:

1. The command parameters were parsed in a protected child environment, guaranteeing that parse errors in our inputs will not affect the main application.

2. Our database has been opened in a protected environment, and cannot be directly affected by the application process except over sockets.

3. The application can affect nothing but its own internal logic.

This protects our application's system environment, but it does not enact protections within. For that, we turn to recent developments in kwebapp and ksql to enforce role access.

In opening the database, kwebapp configure roles as defined in its configuration and passes them into the child process. By leaving access control entirely within a separate process, the web application process is completely restricted from violating its role.

Roles in kwebapp are hierarchical, with child roles inheriting the environment of the parent. (Roles in ksql have arbitrary topologies, keeping instead to the mathematical generalities of RBAC.) Role access is mediated for database routines (insertions, queries, updates, and deletions) as well as data export.

Roles are reentrant toward the parent, such that reentrancy sheds privileges.

In Figure 6, we introduce two types of administrators: administrators for users (user management, `adminusers`), and administrators for other administrators (delegation, `adminadmins`). We define three types of users: buyers (`buyer`), sellers (`seller`), and the combination of buyers/sellers (`buyerseller`).

```
roles {
  role admin {
    role adminusers; # Manage users.
    role adminadmins; # Delegate admins.
  };
  role user {
    role buyer; # Buyers only.
    role seller; # Sellers only.
    role buyerseller; # Buyers/sellers.
  };
};
```

Figure 6: Example role assignment for *foo*.

The administrator sub-types inherit the access of the generic administrator (`admin`) class, while the user subtypes inherit those of the user (`user`).

Roles can be reentrant, as is the case with most sandboxes, where reentrancy must retain or shed privileges, but not augment. So a buyer may shed his or her buying privileges to those of only a generic user.

By default, kwebapp defines three additional roles: the `all` role, which is a common parent to all user-defined roles; the `default` role, which is entered by default when the system begins; and the `none` role, which has no permissions.

While building our kwebapp configuration, we assign roles on the operational level, which maps into SQL statements. As described earlier, this functionality is not available natively in PostgreSQL, which operates on the level of statement types (`insert`, etc.).

By way of example, consider a *user* structure for our application. Using kwebapp, this is rendered as an SQL table (when in SQL output mode), a C structure and function declarations (in C header mode), and the implementation of the functions (C source mode). Several other modes are available beyond the scope of this paper, such as JavaScript classes.

Figure 7 begins by assigning our user a full name, password, and unique identifier. We also stipulate two query types (searching by credentials and by identifier) and the possibility of creating new rows.

We limit access in Figure 7 by setting the access of the `default` role, which describes the condition of our web application upon pre-authenticated access. Common uses of `default` for the user table are usually logging in, which requires queries for credentials, logging out, and insertion of sessions.

The `default` role is our first line of defence, as the

[11] https://man.openbsd.org/pledge.2
[12] https://kristaps.bsd.lv/kcgi

```
struct user {
  field fullname text;
  field hash password;
  field id int rowid;
  insert;
  search fullname, hash: name creds;
  search id: name id;
  role adminusers { all; };
  role default { search creds; };
  role user { search id; };
};
```

Figure 7: Simple role assignments for a user table.

```
struct user {
  ...
  role default {
    search creds;
    noexport;
  };
};
```

Figure 8: Export restrictions (snippet).

application logic is most exposed.

Following the `default` role, we label operations with the `adminusers` and `user` roles as defined in Figure 6. We can also begin with catch-all statements and later narrow permissions, but this is an error-prone approach—it's best to start with proper permissions.

In continuing to build our access restrictions, kwebapp also allows us to build export restrictions, such as in Figure 8. This controls the fine-grained export of data, which is extremely useful to protect against the unauthorised sharing of information.

For example, while an unauthenticated user (in the `default` role) should be able to query the `user` structure, they should never have this information exported. The `noexport` per-role command accomplishes this.

With operation and export restrictions in place, we can take full advantage of kwebapp's current role facilities as in Figure 9.

To guide implementation, kwebapp has an audit mode to illustrate the data model access to any given role. The `kwebapp-audit` utility allows us to view the guarantees given between a role and its data access—the guarantees enforced as described above, with the sandboxed split process architecture.

We can use the JSON output mode of the audit utility to graphically explore the application's roles and their access to all structures. The sample HTML and CSS files given visualise this as a series of tables, colourised and

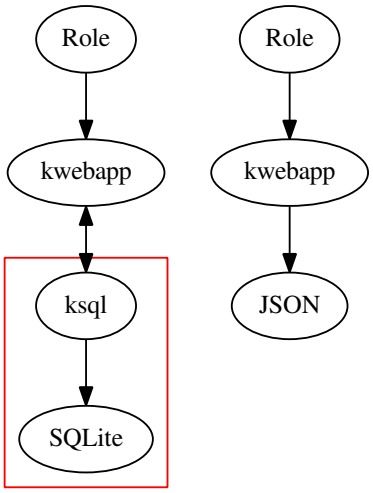

Figure 9: Import and export check against kwebapp roles.

annotated with direct and indirect access, as well as the paths to access.

6 Future Work

While it's clear that ksql should be expanded to support non-SQLite database for future work, there are some larger architectural questions that would strengthen the contract of `noexport` data.

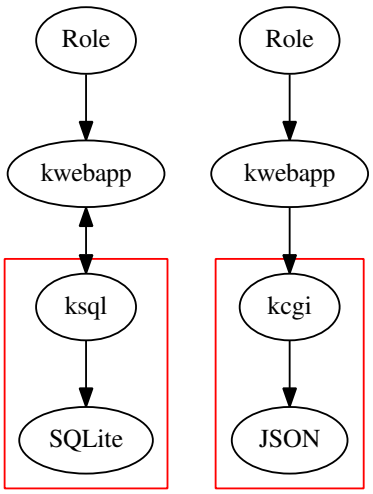

Figure 10: Future work in output restrictions.

In the current implementation, the export routines themselves control for the export of data.

However, recent improvements in kcgi allow for locking output modes, which restricts output to a predetermined set of functions. The primary motivation for this feature is to disallow rogue output within a controlled output stream, such as injecting scripts into HTML output or embedding other executable content. kwebapp can take advantage of this to only allow certain functions to govern output—the same functions that respect the current role.

This can be further strengthened by having the output routines execute from a restricted child process themselves—mirroring the current design of ksql—such as in Figure 10. This would strengthen the export restrictions with hard guarantees.

7 Conclusion

Despite the wealth of materials and implementations supporting RBAC, and despite its clear applicability to web applications, it's surprisingly difficult to find a readily-available end-to-end solution.

However, by exploiting these materials, and by learning from the existing implementations, we're able to leverage the security measures of our modern BSD systems (privilege separation, sandboxing, etc.) to establish guarantees for our roles using kwebapp, which interfaces with the RBAC-enabled ksql.

These tools give us considerable benefits in being able to construct role audits to illustrate that security of our system, as well as potential avenues of exploitation.

8 Acknowledgements

I'd like to thank the AsiaBSDCon secretary and board for accepting this talk and funding travel and accommodation for the duration of the conference. I'd also like to thank CAPEM Solutions, Inc., for their generous support of ksql, kwebapp, and kcgi development toward a safer, auditable solution for security-sensitive systems.

References

[1] BARKER, S., AND DOUGLAS, P. Rbac policy implementation for sql databases. In *IFIP Advances in Information and Communication Technology*, vol. 142. 01 2003, pp. 288–301.

[2] BELL, D. Looking back at the bell-la padula model. In *Proceedings - Annual Computer Security Applications Conference, ACSAC*, vol. 2005. January 2006.

[3] CHANDRAMOULI, R., AND SANDHU, R. Role-based access control features in commercial database management systems.

[4] DE RAADT, T. Mitiations and other real security features. BSDTW 2017, 2017.

[5] DE RAADT, T. Pledge and privsep. EuroBSDCon 2017, 2017.

[6] FERRAIOLO, D., AND KUHN, R. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference* (1992), pp. 554–563.

[7] G. WALTER, K., OGDEN, W., M. GILLIGAN, J., D. SCHAEFFER, D., AND I. SCHAEN, S. Initial structured specifications for an uncompromisable computer security system. 194.

[8] Information technology – Database languages – SQL. Standard, International Organization for Standardization, Geneva, CH, 2002.

[9] LIPNER, S. B. The birth and death of the orange book. *IEEE Annals of the History of Computing 37*, 2 (Apr.-June 2015), 19–31.

[10] PREZ, S., COSENTINO, V., CABOT, J., AND CUPPENS, F. Reverse engineering of database security policies, 08 2013.

[11] SANDHU, R., FERRAIOLO, D., AND KUHN, D. Nist model for role-based access control: Towards a unified standard. In *Proceedings of the ACM Workshop on Role-Based Access Control*, vol. 2000. 01 2000, pp. 47–63.

[12] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. 38–47.

[13] WATSON, R. Introducing supporting infrastructure for trusted operating system support in freebsd. BSDCon 2000, 2000.

[14] WATSON, R., FELDMAN, B., MIGUS, A., AND VANCE, C. Design and implementation of the trusted bsd mac framework. In *Proceedings - DARPA Information Survivability Conference and Exposition, DISCEX 2003*, vol. 1. 05 2003, pp. 38–49.