ACIG APPLIED CYBERSECURITY & INTERNET GOVERNANCE

NASK

Utilizing Object Capabilities to Improve Web Application Security

Received: 24.10.2022
Accepted: 08.11.2022
Published: 16.11.2022

Michael Koppmann SBA Research Vienna, Austria, ORCID: 0000-0001-5699-8226
Christian Kudera SBA Research Vienna, Austria, ORCID: 0000-0003-1772-039X
Michael Pucher University of Vienna, Austria, ORCID: 0000-0003-0123-0214
Georg Merzdovnik SBA Research Vienna, Austria, ORCID: 0000-0002-9955-7284

Cite this article as:
M. Koppmann, C. Kudera, M. Pucher, G. Merzdovnik, "Utilizing Object Capabilities to Improve Web Application Security," ACIG, vol. 1, no. 1, 2022.
DOI: 10.5604/01.3001.0016.0823

Corresponding author:
Michael Koppmann,
SBA Research Vienna, Austria;
Email: mkoppmann@sba-research.org

Abstract

Nowadays, more and more applications are built with web technologies, such as HTML, CSS, and JavaScript, which are then executed in browsers. The web is utilized as an operating system independent application platform. With this change, authorization models change and no longer depend on operating system accounts and underlying access controls and file permissions. Instead, these accounts are now implemented in the applications themselves, including all of the protective measures and security controls that are required for this. Because of the inherent complexity, flaws in the authorization logic are among the most common security vulnerabilities in web applications. Most applications are built on the concept of the Access-Control List (ACL), a security model that decides who can access a given object. Object Capabilities, transferable rights to perform operations on specific objects, have been proposed as an alternative to ACLs, since they are not susceptible to certain attacks prevalent for ACLs. While their use has been investigated for various domains, such as smart contracts, they have not been widely applied for web applications. In this paper, we therefore present a general overview of the capability-based authorization model and adapt those approaches for use in web applications. Based on a prototype implementation, we show the ways in which Object Capabilities may enhance security, while also offering insights into existing pitfalls and problems in porting such models to the web domain.

Keywords

Object Capabilities, secure design patterns, web security

1. Introduction

Exploiting security vulnerabilities for criminal activities has become a business that costs companies worldwide billions of U.S. dollars a year [1]. By 2026, the cybersecurity market size is forecast to grow to 345.4 billion USD [2]. Issues in how the security model of modern web applications are designed form part of this problem. The *Open Web Application Security Project* (OWASP) "Top 10" provides a regularly updated list of the most common problems in web applications, listing *Broken Access Control* as the Top problem in 2021 [3]. This is especially problematic in current web applications, which are commonly built around the concept of *Access-Control Lists* (ACLs). In such systems, authorization and performing a request are distinct actions, and the application itself has ambient authority and power, granted by the underlying operating system. Many techniques and patterns were developed to mitigate risks and security problems that are inherent to this kind of authorization scheme.

In this paper, we present an alternative approach to authorization based on the concept of *Object Capabilities*, which is not susceptible to attacks common in ACL-based systems. By using *Object Capabilities*, the *Principle of Least Privilege* (PoLP) [4] is built-in by design, mitigating the risk of the most common web attacks. A capability is described as a token of authority. It is a reference to an object, including a set of privileges or permissions. This token is transferable and unforgeable [5]. Together with other techniques, this concept can be used to implement complete authorization frameworks.

The aim of this paper is to provide a collection of capability-based security patterns for web applications, which prevent certain classes of vulnerabilities by design. A requirement for those patterns is to provide practical benefits for real-world web applications. Therefore, we focus on maintaining compatibility with the currently existing ecosystem of software products.

In particular, the main contributions of this paper are as follows:

- We provide an overview of capability-based designs in other application domains.
- We utilize these design patterns to illustrate how object capabilities can be utilized to improve web application security.
- With our capability-based reference implementation, *Eselsohr*, we illustrate the feasibility of the design patterns in practice.
- We make the source code of Eselsohr publicly available[1].
- We discuss changes to Browsers that are necessary for full utilization of the benefits of Object Capabilities in web applications.

[1] https://github.com/mkoppmann/eselsohr

2. Background

A key concept in security is the concept of authorization and different approaches try to tackle this problem from different angles. Examples of this include identity-based authorization models revolving around permissions of specific users, or object capabilities granting the permissions itself, without necessarily involving identities.

2.1. Identity-based Authorization Models

The most common identity-based authorization model is the ACL. It's a list of permissions associated with an object. It specifies which subjects are granted access to it and which operations they are allowed to perform. One way to visualize this is as columns in an *Access Control Matrix* [6]. There are other identity-based authorization models, such as *Role-based Access Control* (RBAC) [7] and *Attribute-based Access Control* (ABAC) [8]. For our purposes their differences are not significant, so we will use ACLs as an umbrella term for identity-based access control.

2.2. Object Capabilities

A capability is a *transferable* and *unforgeable* token. It contains a reference to an object or a resource and the set of allowed actions that can be performed on it. Because the word "capability" is so overloaded with different meanings and because this concept overlaps with many object-oriented programming principles, nowadays, the term *Object Capabilities* (OCAP) is generally preferred. Nonetheless, capabilities are not tied to object-oriented programming and have been in use in various different contexts, first being mentioned in 1966 while discussing concurrent programming [9], in addition to being the basis for sophisticated implementations of operating systems [10, 11], hardware [5], kernels [12], file systems [13], and more. Because capabilities combine both designation and authority—meaning that they specify a particular resource and what access is allowed—whoever possesses a capability can exercise its authority. One of the key properties of capabilities is the ability to *transfer* them. *Unforgeability* is another important aspect of capabilities. It guarantees that they can only be accessed via (I) *creation*, (II) *transfer*, and (III) *endowment* [14]. Capability-based systems follow the *Principle of Least Authority*[2] (PoLA), as it follows from using object capabilities. This allows for collaboration between untrusted parties, as the potential damage that can be caused by a malicious actor is reduced to a minimum.

[2] We use authority and privilege interchangeably in this work, although subtle differences exist.

3. Related Work

In the context of web applications, ideas for capability-based security are often modelled upon already established standards. For example, *Bearcaps* [15, 16] and *Bearer URL* [17] both rely on URI schemes to represent capability tokens. Bearcaps are URIs with two parameters: The access token and the web URL, while Bearer URLs rely on a similar syntax to HTTP Basic Authentication URLs. Compliant browsers should not reveal the token parameter through UI elements/JavaScript and compliant web servers should not log the tokens. While these URI schemes could be modelled through JavaScript and the HTTP Authorization header, extensive browser and server support is needed to protect the Bearer tokens against cross-site scripting (XSS) attacks. Instead of building on URI schemes, *Macaroons* [18] make use of cookies, extending them with caveats that attenuate or confine them and thus rendering them more suitable for authority delegation purposes. Caveats are nested and chained HMACs, used to append restrictions to the cookies, restricting usage and marking the need for additional authentication proofs. Each appended caveat consists of a list of predicates and a request is only allowed if all of these predicates are fulfilled.

The *Grant Negotiation and Authorization Protocol* (GNAP) [19], formerly known as OAuth 3.0, is an in-progress next generation protocol tackling the authorization problem from a different angle. While it is based on the experience of implementing OAuth 2.0 in practice, it is not compatible with previous OAuth standards. GNAP supports features like a built-in concept for identities and the ability to differentiate between running instances of the same app. GNAP is a protocol that can work with the *Authorization Capabilities for Linked Data* (ZCAP-LD) [20] data model. ZCAP-LD combines object capabilities and *Linked Data Proofs* [21] to allow delegating authority in a distributed network by chaining together capability documents. Each document can be further restricted by adding caveats to them to restrict their scope, their lifetime, or to revoke them later on.

While these projects work on introducing capability-based authorization models, they do not make direct use of OCAP themselves. The OCAP community continues to improve their security model steadily, working on projects beyond web applications and collaborating with committees to standardize their techniques. One of the more recent approaches is *Endo* [22], a sandboxed and OCAP-safe subset of JavaScript. It offers protection against malicious third-party dependencies by explicitly locking privileged features, e.g. network interaction, behind capabilities. Reviewable policies are used to restrict the authority of dependencies to a minimum. As part of the overall project, a new ECMAScript standard called *ShadowRealm* is currently proposed, which would dampen the impact of XSS attacks [23] by running user-provided input in an environment void of capabilities.

Other OCAP applications include *The Spritely Project* [24], which uses object capabilities to build a platform for federated social networks. The *CapTP* protocol, the foundational layer for Spritely, enables OCAP in distributed computing and supports different transport layer protocols, e.g. peer-to-peer applications through Tor [25]. In a similar manner, *Cap'n Proto* [26] is a serialization format and RPC framework, which includes OCAP-based security as one of its core principles. On a lower level, *Fuchsia* [27] is an operating system developed by Google, which prioritizes security as one of its design goals from the start. It makes use of OCAP, by enforcing all system calls to go through their defined virtual *Dynamic Shared Object* (vDSO) interfaces, allowing more fine-grained access control through capability handles, when compared to typical system call ABIs.

4. Capability-Based Design Patterns

A capability in the context of programming languages is a reference to some piece of data. Although in object-oriented languages this is usually a reference to an object, it is not in fact limited to that. A capability can also be a reference to a primitive type, a function, a closure, or other data types, and can therefore be used with other programming paradigms, such as functional programming. In order to guarantee unforgeability, the programming language being used has to support "safe" pointers. References are pointers to a specific address in memory where data are stored. The language must protect these, such that it is not possible to, for example, cast an integer to a pointer (as is the case for the C language). Only creation, transfer, and endowment should give access to object references.

Transferability can be given by the fact that data, or references, can be passed as arguments to functions. In general, OCAP-based development can be achieved by omitting global scope, passing arguments, and utilizing lexical scoping [28].

```
function mkCounter() {
 let count = 0;

 return Object.freeze({
  increment: function () {
   return count++;
  },
  decrement: function () {
   return count--;
  }
 });
}

counter = mkCounter();
entryGuard.giveCounter(counter.increment);
exitGuard.giveCounter(counter.decrement);
```

Listing 1: Counter example in JavaScript showing OCAP programming

Listing 1 demonstrates a "counter" example in JavaScript [29]. The function `mkCounter` contains a mutable variable called count and returns a JavaScript object that contains two functions: one to increment the `count`, and one to decrement it. `Object.freeze` creates an immutable version of the object. The `count` variable is only accessible within this function, a closure, and the only way to manipulate it is by calling one of the two provided functions, since they have access to the variable, being in the lexical scope. This is equivalent to a class in object-oriented languages where `mkCounter` is the constructor, count a private instance variable, and `increment` and `decrement` the public API. Lines 14–16 illustrate how to use this as a security mechanism. Imagine Alice has access to two other objects called "entryGuard" and "exitGuard". The entry guard should only have the power to count up, while the exit guard should only count down. Alice creates a new counter object and passes only the `increment` function to the entry guard and only the `decrement` function to the exit guard. This is also a demonstration of PoLA, as both guard objects are only being given the authority they need to do their job. Similar to design patterns in object-oriented design, several patterns emerged for programming with object capabilities. We have collected some of these patterns and adopted them for use within the web security context. The following sections describe four of these patterns in detail.

4.1. Revoker

Listing 2 shows an example of the *Revoker* pattern [29]. When Alice passes an object reference to Bob, she has no means of forcing the reference to be returned. Early research work assumed that this limitation is a downside of object capabilities [30]. The Revoker pattern demonstrates how revocability can still be provided in an OCAP system. The `mkRevocable` function takes a function as an argument and returns an object with two functions: `wrapper` and revoke. The `wrapper` function can be passed to other objects, in this example to Bob, who can then proceed to interact with the wrapped function. If Alice later regrets that decision because Bob started to act strange, she can call the `revoke` function, which she has kept to herself. Then, `revoke` will set the function in the closure to `null`, rendering all further requests to the wrapper by Bob unusable.

```
function mkRevocable(fn) {
  return Object.freeze({
    wrapper: function (...args) {
      return fn(...args);
    },
    revoke: function () { fn = null; }
  });
}

revokableFoo = mkRevocable(foo);
bob.bar(revokableFoo.wrapper);
revokableFoo.revoke();
```

Listing 2: Revocation example in JavaScript

4.2. Membrane

Listing 3 shows an example of the *Membrane pattern* [31]. This can be used at the boundaries of the program architecture, where input and output with the real world has to be provided. Membranes then can be used to reduce the authority within the program by limiting the surface of available powerful capabilities. The function `mkReadOnlyFile` takes a file object as an argument and returns a new object that only exposes a subset of the original available functions.

```
function mkReadOnlyFile(file) {
 return Object.freeze({
    read: file.read,
    getLength: file.getLength
  });
}
```

Listing 3: Membrane example in JavaScript

4.3. Sealer

Listing 4 shows an example of the *Sealer* pattern [32, 33]. This can be used to securely transfer data between multiple objects without revealing the content to everyone involved. Similar to the Counter example, the `mkSealer` function uses `sealed` as its private state. The variable `sealed` is a WeakMap, a key/value store, where objects are keys; it is also not enumerable. It returns an object with two functions: seal and unseal. The seal function expects an argument called `data` and creates an empty object called box. The object identity of box is then used as key for `sealed` and `data` is used as value. The variable box is then returned to the caller. The function unseal takes a box as an argument and uses it to extract the value from the map.

Alice passes a `sealer` to Bob, which he can then use to send Alice a secret. Bob does not have a reference to Alice but Carol has. Since Bob has a reference to Carol, he sends her his box called `secretForAlice`. Carol, having no access to the unseal function, cannot see which secrets are being passed around and sends the box to Alice. Alice can access the secret by calling `unseal` with the provided object key.

```
function mkSealer() {
  let sealed = new WeakMap();

  return Object.freeze({
   seal: function (data) {
      const box = {};
      sealed.set(box, data);
      return box;
    },
    unseal: function (box) {
      return sealed.get(box);
    }
  });
}

// Alice
sealer = mkSealer();
bob.foo(sealer.seal);

// Bob
secretForAlice = sealer.seal("Hunter2".);
carol.bar(secretForAlice);

// Carol
alice.baz(secretForAlice);

// Alice
secretFromBob = sealer.unseal(secretForAlice);
```

Listing 4: Sealer example in JavaScript

4.4. Limited Use

Listing 5 shows an example for the *Limited Use* pattern. This is a variation of the Revoker pattern, where we restrict the longevity of object capabilities. Instead of having an explicit `revoke` function, it has a built-in counter state that is reduced by one each time the wrapped function is called.

```
function mkLimitedUse(numOfInvocations, fn) {
  let usages = numOfInvocations;

  return Object.freeze({
    use: function (...args) {
      if (usages > 0) {
        fn(...args);
        usages--;
      } else {
        return;
      }
    }
  });
}
```

Listing 5: Limited Use example in Javascript

These patterns also compose well together; to create a one-time use, read-only, revocable file capability, these constructs only have to be stacked: `mkLimitedUse(1, mkReadOnlyFile(revokableFile))`. In the end, it all boils down to two guiding principles that allow a reduction of authority in program development and the risk of intentional or accidental bad behaviour:

1. No usage of global mutable state. This enforces that mutable data must be passed explicitly between objects, allowing controlling the flow of authority, thus, reducing the amount of potentially malicious actors in a system, who can manipulate a given piece of information.
2. Only controlled communication with the outside world. Interactions involving input and output should be wrapped at the edges of the program's architecture, providing a safe subset of possible functions. This allows that OCAP rules can be enforced within the program itself, while possibly dangerous code sections stay small and auditable.

These principles are sometimes easier, sometimes harder to follow, depending on the programming language and tooling in use.

5. Eselsohr

To illustrate the feasibility of our introduced capability-based design patterns in practice, we implemented Eselsohr, a bookmark manager where URLs can be stored in collections, which can then be shared with other people. Eselsohr supports the following features:

- **No Requirement for User Accounts.** Eselsohr does not require users to register before they can use the web application. Performing privileged actions is done by providing access tokens in URLs.
- **Support for Multiple Collections.** A single Eselsohr instance can handle multiple collections without any concept of an account. Access is granted by authorization, which does not require authentication.
- **Shareable Permissions.** Access to a collection is granted with URLs. New URLs with reduced permission sets can be created by users. Links can expire or be revoked by their owners.
- **Simple Embeddability.** Privileged Eselsohr actions can be integrated into other web applications. This is possible because the designation of a resource is coupled with the authority to perform the action.

We highlight the applicability of the design patterns across languages, by implementing Eselsohr in Haskell, a statically typed, immutable by default, purely functional programming language with lazy evaluation [34]. Although Haskell is a functional programming language and lacks the concept of an object, while also favouring immutability, many object capability patterns can still be applied with certain modifications.

As object capability languages also try to achieve functional purity [35], Haskell's strictness on the separation between values and effects fulfils this goal. Eselsohr uses types in two ways as a kind of capability for achieving the principle of least authority: as access tokens within the runtime and to limit the possible effects it can have. The following sections describe how a capability-based approach is used in different layers of the application to provide features, improve maintainability, or enhance security. The goal is to reduce the ambient authority and stick to the principle of least authority.

5.1. Types as Capabilities

In short, external authorization systems like OAuth2 work like this:

3. A client wants to access a resource. They must prove that they are authorized to do so.
4. The client presents claims, such as their identity and the requested scope, to an authorization service.
5. This service performs the necessary authorization checks and returns a cryptographically signed token to the client.
6. The client presents this token to the resource service, which verifies the validity of the token before the service allows access to the requested data.

Types can be used to simulate this behaviour without using any cryptography but secured by the runtime of the language [36]. This can be achieved by using types with private constructors, which are functions that can create values of that specific type but are not exported outside of their respective module. Other modules, therefore, cannot create values of that type directly but have to use the exported constructor function. Within this private constructor, all necessary authorization checks can be performed.

```haskell
module Authorization
  ( AccessToken
  , getData
  , AccessArticle(..)
  , DeleteArticle(..)
  , accessArticleToken
  , deleteArticleToken
  ) where

{- import required types and functions -}

-- | Constructor of the 'AccessToken'. type
newtype AccessToken a = a

-- | Function to access the wrapped value
getData :: AccessToken a -> a
getData (AccessToken data) = data

data AccessArticle = AccessArticle Id

data DeleteArticle = DeleteArticle Id

-- | Authorization function that maybe returns
-- the requested accesstoken or nothing, depending
-- if the checks succeed or not.
accessArticleToken :: Id -> User -> Maybe (AccessToken AccessArticle)
accessArticleToken articleId principal =
  if {- perform authorization checks -}
    then Just (AccessToken (AccessArticle articleId))
    else Nothing
```

Listing 6: Example authorization module

```
-- | Same as accessArticleToken but with different
-- checks
deleteArticleToken :: Id -> User -> Maybe (AccessToken
DeleteArticle)
deleteArticleToken articleId principal =
  if {- perform different authorization checks -}
    then Just (AccessToken (DeleteArticle articleId))
    else Nothing
```

A privileged function, such as a database-accessing one, would not then accept plain values as arguments but values that are members of such authorization types. These types can be unwrapped to access the required argument to perform the requested action. Values of such types work as a proof that the required authorization check has happened in the past as immutability guarantees that no change could have happened in between. Listing 6 shows such an authorization module, which handles access tokens. The type AccessToken wraps a generic type a, whose constructor is not exported from the module. The exported function `getData` can be used to unwrap the contained value. The types `AccessArticle` and `DeleteArticle` represent permissions to access or delete articles from the database respectively. The function `accessArticleToken` expects an article Id and a User as an argument and then performs authorization checks. If it succeeds, a value with the type Maybe (`AccessToken AccessArticle`) is returned. The same happens for deleteArticleToken with Maybe (`AccessToken DeleteArticle`). The caller of those functions can then decide how to continue, depending on the result.

```
module Database where

{- import required types and functions -}

getArticle :: AccessToken AccessArticle -> IO Article
getArticle token = do
  let (AccessArticle articleId) = getData token
  getArticleFromDB articleId

updateArticle :: AccessToken AccessArticle -> Article -> IO ()
updateArticle token updatedArticle = do
  let (AccessArticle articleId) = getData token
  updateArticleFromDB articleId updatedArticle

deleteArticle :: AccessToken DeleteArticle -> IO ()
deleteArticle token = do
  let (DeleteArticle articleId) = getData token
  deleteArticleFromDB articleId
```

Listing 7: Example database module

Listing 7 shows how a module with privileged functions can use these access tokens to guarantee that the caller performed an authorization check. The functions getArticle and updateArticle are expecting a value of type `AccessToken AccessArticle` instead of a plain Id. Therefore, it is not possible to call this function without a preceding call to `accessArticleToken`. In `deleteArticle` a different authorization check is enforced by requiring a different type of access token.

```
unauthorizedAccessToken :: a -> AccessToken a
unauthorizedAccessToken perm = AccessToken perm
```

Listing 8: Creation of access tokens without authorization

Some use cases require the unauthorized call of privileged functions, such as the initial fetching of the user value, as it is required for performing the authorization checks. Listing 8 shows a function which takes an argument with a generic type and converts it into the `AccessToken` type. The prefix `unauthorized` serves as a hint and can be detected during code reviews or by automatic tooling. This technique cannot merely be used for authorization but also for validation of external data. For example, instead of representing an email address as a `String`, a specialized `Email` type can be created. The constructor of that type guarantees that it follows a specific pattern, such as containing an @ symbol. We can further split this type up into two separate types, representing a verified email and an email that the user has yet to verify. Functions that expect a verified email have static guarantees that the verification step has been performed.

Further examples include the differentiation between SQL queries and data through the use of types, so that user input cannot be accidentally concatenated; thus preventing SQL injection attacks. Another example is the enforcement of proper output encoding in an HTML layout engine to prevent cross-site scripting attacks, by requiring that user input has to be converted to an HTML type.

Using types that enforce invariants and which represent concepts in the domain of the application is a pattern also known as "value object" in the realm of domain-driven design [37]. These types are implementations of the Wrapper pattern. Working on primitive, built-in types like `String, Bool`, or Integer allows for too much latitude within the application. They are missing restrictions and a proper guidance for developers, which leads to a more fragile architecture. Utilizing the type system as a quasi-state machine, specifying the direction in which data has to flow through the application, enables the development of a more formal application design.

5.2. Types for Explicit Side-effects

The IO type gives us too much ambient authority. Adhering to PoLA means that we want to reduce the possible effects to a minimum. Object capability languages, such as Monte, only allow the import of IO-providing functions at the entry point of a module [38]. These functions are then passed along as arguments until they get called. This reduces the number of side-effects to the ones declared at the entry point. Eselsohr achieves a similar explicitness by using a custom data type that contains IO and Haskell's type class system [39]. Haskell's mechanism to generalize behaviour and patterns over multiple data types is called a type class. Examples of other type classes are `Eq,` for checking equality; Ord, for checking the ordering of elements; or `Num`, for numeric operations. For our purposes, type classes can be seen as similar to interfaces in object-oriented languages. Listing 9 shows the type class of `Eq`, which includes the equal (==) and not equal (/=) functions.

```haskell
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Listing 9: Type class of Eq

Listing 10 shows the type signature of a polymorphic function called `uniq`, which, as for the Unix command-line tool, removes repeated adjacent lines in a list of a's. It works with any type a as long as the type has an implementation of the `Eq` type class. To reduce the number of possible side-effects in our program, and to simulate the approach taken by object capability languages, Eselsohr uses a custom data type and type classes to explicitly declare all possible side-effects per function.

```haskell
uniq :: Eq a => [a] -> [a]
```

Listing 10: Type signature of a polymorphic function with Eq constraint

Functions that work with side-effects use—instead of running in IO directly—a generic type m. This type is then constrained by type classes that represent effects. The business logic of the application therefore remains polymorphic, and can be either completely pure or emit effects, depending on the data type that implements those type classes. This custom data type is called App in Eselsohr. Listing 11 shows such a function. Here, createArticle takes a Uri as an argument and returns an Article in the polymorphic type m. This type is constrained by having an implementation for the classes MonadScraper and MonadTime. The first class provides the function scrapWebsite, while the second class provides currentTime.

```haskell
createArticle :: (MonadScraper m, MonadTime m) => Uri -> m Article
createArticle uri = do
  aTitle   <- scrapWebsite uri
  aCreated <- currentTime
  pure (Article aTitle uri Unread aCreated)
```

Listing 11: Example function showing effect type classes for a custom monad

Listing 12 shows the implementation of the MonadTime class for the App monad, which uses an IO function from the *time* package. MonadIO is another generalization and requires our App monad to have the ability to run IO actions. Only a thin, auditable layer of pure IO functions now wraps the logic of our application. Environment variables are parsed into a configuration data structure, folders in the file system are prepared, the App monad is created, and the web server is started, which executes our application logic for every incoming request. As type classes turn into dictionaries with functions as values at compile time, which are then passed along implicitly [39], we simulate the behaviour of object capability languages like Monte, where passing side-effecting functions happens explicitly.

```haskell
import Data.Time (UTCTime, getCurrentTime)

class (Monad m) => MonadTime m where
  currentTime :: m UTCTime

instance MonadTime App where
  currentTime = currentTimeImpl

currentTimeImpl :: MonadIO m => m UTCTime
currentTimeImpl = liftIO getCurrentTime
```

Listing 12: Example effect class representing access to the time packagemodule

This is a form of the Sealer pattern. Only functions that were given a specific type constraint can access the IO functions contained in the App monad. Other functions could accept IO functions as arguments or return them, thus enabling them to pass these capabilities through the application. However, without the necessary type constraints they are not powerful enough to execute the capabilities. We achieve a clear separation between pure functions, which cannot cause any harm, and actions having effects, which should be the primary target when conducting security audits.

5.3. URLs as Capabilities

Eselsohr uses web-keys to transmit access tokens over HTTP, which was a compromise in the design process. The web-key technique [40] is an access control technique developed within the context of the Waterken web server, which in turn is written in an OCAP subset of Java. With web-keys, every capability is assigned a different 64-bit string, which is passed alongside URLs in a Base32-encoded form. It is used within Eselsohr as a means of passing capability-tokens without involving JavaScript, but special care is needed to prevent the token from leaking in the HTTP Referer header.

```
https://eselsohr.example.org/articles/example-title?acc=QMANQJKQCLCDX
JT5DI[...]VIPHU52S72SPX2CI2GU
```

Listing 13: Example URL to access a single article resource

Listing 13 shows how a single article resource can be accessed via a capability URL. The acc query parameter is a Base32 and binary encoded Haskell data type containing the ID of the referenced file and the capability. Eselsohr does not have a concept of users; article collections are stored in separated files and are identified by the access token. Alternatively, two separate query or path parameters could be used to avoid the need for serialization. The referenced capability has an optional expiration date, set to one month by default, and contains a reference to a resource, like an overview page or a single article, including a set of permissions. When an endpoint is called, types enforce the checking of these permissions before any resource can be loaded.

Users can create new URLs for each page with restricted permissions and expiration dates, implemented with a combination of the Membrane[3] and the Limited Use pattern. This makes it possible to create, for instance, time-limited read-only or append-only access to certain resources. Working with fine-grained permissions allows for dynamic use-cases that are hard to implement in static, group-based, coarse-grained scenarios. The generated URLs can also be revoked at any time, giving the owner full control over the access management by using the Revoker pattern.

[3] A variation of the Membrane pattern had to be used, where a list of permissions is used, instead of embedding functions directly, as the capabilities had to be serializable for persistence.

The initial capability given to a person after creating a new collection is the entry point to the application. People are encouraged to store this URL somewhere safe, such as inside a password manager. This link to the initial overview page replaces the login page, since people do not have to authenticate themselves in order to use Eselsohr.

Omitting a central login page grants Eselsohr another property: its resistance to phishing attacks. By combining the designation and authority into a URL, transmitted over an encrypted and authenticated HTTPS tunnel, the user agent has the burden of verifying the authenticity of this connection. Traditionally, with usernames and passwords, the user is responsible for identifying whether the login form belongs to the right actor.

6. Evaluation

To reliably evaluate our proposed capability-based design patterns, we compare conceptual differences between Eselsohr and the open-source alternatives *Wallabag* [41] and *Espial* [42], both built on identity-based security paradigms. Wallabag was started in 2013 and is written in PHP. It is built on the Symfony [43] web framework. Wallabag can extract the content of web pages and display it in a more user-friendly format. Several import and export functions are available that aid migration to or from the service. Espial was started in 2019 and is written in Haskell and PureScript. The backend is built on the Yesod [44] web framework, while the frontend is written in PureScript [45], a Haskell-like language that transpiles to JavaScript and which is used to build *Single Page Applications* (SPAs). Espial users can also add, in addition to web pages, notes with support for Markdown. Articles cannot only be added by browsing to the web application and using the corresponding form, but also by using a *bookmarklet*, a JavaScript snippet that can be bookmarked.

For the evaluation, we compare four main aspects of web applications: (i) user management, (ii) data manipulation, (iii) resource sharing capabilities, and (iv) embeddability and integration features.

6.1. User Management

All identity-based web applications require some kind of user or account model, which is then used for authentication and authorization. Additionally, secure password hashing algorithms, brute force protections, and session management are also needed. These common functionalities are often provided by the web framework in use. Both of the identity-based apps under consideration have built their user models on top of code provided by their chosen frameworks.

The OWASP list associates several risks with user management, including: (I) storing passwords in plain text, in encrypted form, or with a cryptographically weak hashing algorithm; (II) allowing weak or well-known passwords; (III) implementing a vulnerable password reset; (IV) missing brute force protections for the login procedure; (V) missing multi-factor authentication.

Because implementing secure user management is a non-trivial task, there is a trend in the industry of delegating this to third-party providers and using *Single Sign-On* (SSO) solutions like *SAML* or *OpenID Connect* for authentication and authorization [46]. Of course, this also increases the risks associated with centralization. If the same account, hosted by an identity provider, is used for a multitude of different services and access to it is removed temporarily or permanently, then either these applications can no longer be used or else the user has to start over with a new account.

With object capabilities, the concept of identity is optional. OCAP-based applications, like Eselsohr, do not require implementing user management and can therefore avoid the complexity and potential security issues associated with it.

6.2. Data Manipulation

Typically, identity-based web applications perform data manipulation with operations commonly known as "Create, Read, Update, Delete" (CRUD). For example, adding a new bookmark consists of:

1. The web application accepts a new request from the user.
2. A routing mechanism maps the URL path from the request to a controller, which handles requests for that particular path.
3. Inside the controller, data from the URL and HTTP body are optionally extracted if needed.
4. Some kind of authorization check is called to verify whether the calling user is allowed to perform the action.
5. Data are validated when handling user input, where it has to meet some criteria for further processing.
6. The controller calls a service performing the business logic or performs the logic itself; this typically involves a database.
7. Based on the return values of the service, a response is sent back to the user.

Checking if a subject is allowed to call a particular function or endpoint is called function-level authorization. Verifying that a subject is allowed to access a specific object is called *object-level* authorization. Validation of both of them for every single access is also called the Principle of *Complete Mediation* [47].

In Espial the function `_handleFormSuccess`[4] handles the main part of the controller's logic. It receives HTML form data from the user and starts by requiring that the current user has to have a valid authenticated session by calling `requireAuthPair`. This authorization function is provided by the used web framework. In this part of the application, object-level authorization is not used, as every user is implicitly allowed to add new bookmarks. The rest of the function then performs data validation, stores the new bookmark in the database, and archives the content of the bookmark. The main problem with this kind of controller logic is that the authorization logic is optional.

[4] https://github.com/jonschoning/espial/blob/c3a126b9eadb3c3778ab93ed4c4d0e80%35f669d3c/src/Handler/Add.hs#L59

There is no enforcement of access restriction to that endpoint or on objects themselves. In this case, the function `requireAuthPair` is also used for obtaining data about the currently logged-in user, so it is unlikely that this function be overlooked, yet there are other controllers where that could be the case. In addition, if Espial chooses to add a less privileged user group, which does not have permission to create new bookmarks, new authorization checks would then have to be added, without any guidance by the compiler or framework.

An OCAP-style web application, like Eselsohr, solves these problems on an architectural level. Access checks are statically enforced by embedding the result of authorization checks in type-level access tokens. Service code can then require such tokens, guaranteeing that authorization was successful. By using web-keys, it is not possible to designate a resource without the associated permission set, so we always fulfil the complete mediation principle.

6.3. Sharing Functionality

When using a web application, users have certain expectations compared to traditional desktop applications, such as the ability to share links to web pages or to bookmark them [40]. Identity-based applications usually only have the choice between public pages, which can be accessed by anyone, or private pages, which require users to be logged-in when they open the link, since designation and authority are split.

This is the case in Espial. The bookmarks of users are public by default and are available at https://espial.example.org/u:username/. They also have the choice to declare a bookmark private, which hides it from that user's public page and requires an authenticated session. There is no functionality to share pages with a limited set of other people.

In Wallabag bookmarks are private by default, but they can be put into a public, read-only mode. In addition, *unread*, *archived*, *starred*, or *all* articles can be shared over RSS feeds. For this to work, Wallabag generates a 14-character-long random token as part of the feed's URL path, which acts as a capability, and which can also be revoked by the user at any time. It is not possible to generate multiple tokens or to place further restrictions upon them. The same token is also used for all available feeds, but it is still a very basic capability system, embedded in an otherwise identity-based application.

Web applications built on object capabilities, and web-keys specifically, take this concept further and allow everything to be shared with links if desired. For example, Eselsohr allows applying further restrictions on web-keys, such as a limited validity period and restricted permission sets.

6.4. Embeddability

The inability to embed identity-based products is also a weak point of them. HTML provides the functionality to embed other web pages with the `iframe` element, but its usefulness is often diminished for fear of security vulnerabilities. These frames are the main attack vector for clickjacking attacks and the main prevention method is by disabling the option to embed a web site completely, or at least for cross-origin requests. Once again, this works because an attacker can link to a well-known endpoint from a popular web site on their attack page and trick other people to reveal sensitive information or perform authorized actions, because the ambient authority granted by their browsers cannot differentiate between benevolent or malicious intent. This hinders the ability to build collaborative web applications.

Object capabilities allow for secure embedded pages and collaboration. Clickjacking, a confused deputy attack, is no risk for OCAP-based web applications, as an attacker would need to know the web-key for the page they want to link. At this point, they would already have access and have no need for social engineering techniques. Such applications can safely omit the HTTP headers that disallow framing the web site and allow other web applications to embed them as they like.

As an example, Eselsohr's new article functionality could be added as a custom widget in the instant messenger Element [48]. This messenger enables the embedding of arbitrary web pages as iframes by providing a link to them. Usually, only pages that do not require authorization can be used for this, as cookies cannot be used to show everyone in the channel the same page. However, web-keys do not have this limitation and such scenarios are therefore permitted. The provided web-key only has the permission to create new articles for that specific resource and nothing else. If the link leaks, no other actions could be performed with it.

6.5. Discussion

The above comparison between Eselsohr and two identity-based projects shows that security vulnerabilities in authorization systems can be prevented with capabilities by design. By using types as authorization tokens, we obtain strong guarantees that authorization checks will not be overlooked and that services cannot be called unauthorized. Web-keys, a combination of designating a resource and a corresponding set of permissions, are not susceptible to confused deputy attacks and are therefore resistant to vulnerabilities like cross-site request forgery or clickjacking. Using the type system for explicit side-effects enhances the reasoning about the code base, since it provides a better understanding of which functions are safe to call and which are potentially dangerous. In addition, by disallowing arbitrary side-effects everywhere, certain areas of the program, such as those that handle untrusted user input with deserialization, become secure. In identity-based systems, subjects cannot choose which authority they want to use when accessing a resource. Authority is implicitly available in the environment and is granted based on the identity of the caller. In the context of code, this means that every function can potentially perform any action, as all code possesses equal authority. By requiring access tokens within the code, and thus making authority explicit, the authorization flow in the program becomes equivalent to the creation and passing of access tokens as arguments. Functions have to explicitly request authority before they can be used.

As subjects in identity-based systems do not have explicit control over authority, they cannot declare a purpose when accessing a resource. Therefore, a subject cannot securely perform actions on behalf of others, as all actions will use the authority of the subject. Object capabilities, on the other hand, combine designation with authority, thus allowing for the use case stated above. Collaboration is secure, since the subject is able to use each capability for its intended purpose. To uphold the principle of least authority, we want to grant subjects the minimum required amount of authority they need to perform their tasks. This can be done in an identity-based system by creating small identities with minimum rights, though it is hardly practical.

7. Conclusion

In this work, we proposed an alternative authorization model for web applications, utilizing object capabilities: object references combined with an associated set of permissions. We showed which security vulnerabilities arise when designation and authority are split apart in the context of web applications, and how this problem is inherent to applications built on access-control lists. In the addressed scenarios, we were then able to demonstrate that programming in an object capability style helps to eliminate certain security vulnerability classes on an architectural level. Alongside this, we provided some techniques and patterns based on this style, such as web-keys.

A functional prototype was implemented to demonstrate that these techniques could be used in practice. The security analysis and model evaluation showed that OCAP-based applications have no significant drawbacks when compared to ACL-based applications, while providing improvements in areas such as shareability and embeddability. This was done by conducting a security evaluation, with a focus on the most common vulnerabilities found in web applications, and by comparing the prototype with other existing applications. We also looked at how modern browsers can securely exchange data between server and client, and which extensions are needed to better integrate and protect capabilities in web applications. The biggest problem remains that of transferring capabilities over URLs. Although hyperlinks are the primary method of navigating between web applications, web browsers currently assume that URLs only contain non-sensitive information, making it hard to embed sensitive information such as capabilities. In addition, a capability in a URL is a plain string with no further protection, so anyone could come up with a potentially valid capability, even though it was not passed to them explicitly.

The prototype was developed in a programming language that was not explicitly designed for this style of programming. It is capable of being run on common operating systems without the need for specialized application frameworks. As these existing systems are not following OCAP principles and assume ambient authority, adapters and wrappers are required to integrate them into an object capability application.

Finally, an overview of current OCAP-related projects was given, together with recommendations on how the prototype could be further improved in the future.

7.1. Future Work

To overcome the existing drawbacks, future research should evaluate how existing features in web browsers could be used to circumvent the current limitations in regard to transferring capabilities. Eselsohr made use of web-keys—capabilities in URLs—to navigate between web pages, because they work without JavaScript and can be used across different web applications. With JavaScript more methods of transfer would be available, such as adding HTTP headers, non-HTML HTTP bodies, or WebSockets. These channels could then be used to securely transfer capabilities within the same web application.

It would also be of interest to examine how different application architectures effect the effectiveness of object capability security. The implemented prototype utilized static type checking and a monolithic architecture, allowing it to apply techniques that are not available in a dynamically typed language or in a microservice architecture. These design decisions would then require a different set of OCAP-based techniques.

REFERENCES

[1] K. Smith, A. Jones, L. Johnson, L. Smith, "Examination of cybercrime and its effects on corporate stock value," *Journal of Information, Communication and Ethics in Society*, vol. 17, no. 1, pp. 42–60, 2019.

[2] K. Mlitz. (2021). *Size of the cybersecurity market worldwide, from 2021 to 2026*. [Online]. Available: https://www.statista.com/statistics/595182/worldwide-security-as-a-service-market-size/. [Accessed: Oct. 24, 2022].

[3] Inc. OWASP Foundation. (2021). *OWASP top ten 2021* [Online], Available: https://www.hhs.gov/sites/default/files/owasp-top-10.pdf. [Accessed: Oct. 24, 2022].

[4] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[5] H. M. Levy, *Capability-based computer systems*. USA: Butterworth-Heinemann, 1984.

[6] B. W. Lampson, "Protection," *SIGOPS Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974.

[7] D. Ferraiolo and R. Kuhn, "Role-based access control," *15th NIST-NCSC National Computer Security* Conference, 1992, pp. 554–563, doi: https://doi.org/10.48550/arXiv.0903.2171.

[8] V. Hu, D. Kuhn, D. Ferraiolo, J. Voas, "Attribute-based access control," *Computer*, vol. 48, pp. 85–88, 2015, doi: 10.1109/MC.2015.33.

[9] J. B. Dennis, E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[10] J. S. Shapiro, J. M. Smith, D. J. Farber, "EROS: A fast capability system," *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999, pp. 170–185.

[11] A. S. Tanenbaum, M. F. Kaashoek, R. V. Renesse, H. E. Bal, "The amoeba distributed operating system – a status report," *Computer Communications*, vol. 14, pp. 324–335, 1991.

[12] K. Elphinstone, G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 133–150, doi: 10.1145/2517349.2522720.

[13] Z. Wilcox-O'Hearn, B. Warner, "Tahoe: The least-authority filesystem," *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, 2008, pp. 21–26, doi: 10.1145/1456469.1456474.

[14] C. Morningstar. (2017, May 7). *What are capabilities?* [Online]. Available: http://habitatchronicles.com/2017/05/what-are-capabilities/. [Accessed: Oct. 24, 2022].

[15] A. Conill. (2019). *The bearer capability URI scheme*. [Online]. Available: https://git.sr.ht/~kaniini/draft-conill-bearcapsuri-scheme/tree/22c458a95992e56ac41f1fff745855b14a811046/item/draft-conill-bearcaps-urischeme.txt. [Accessed: Oct. 24, 2022].

[16] C. Lemmer-Webber. (2019). *Bearcap URIs*. [Online]. Available: https://github.com/cwebber/rwot9prague/blob/908c5522720f0e3debad2c1578c28a984660ba05/topics-and-advancereadings/bearcaps.md. [Accessed: Oct. 24, 2022].

[17] N. Madden. (2021). *Towards a standard for bearer token URLs*. [Online]. Available: https://neilmadden.blog/2021/03/20/towards-a-standard-for-bearer-token-urls/. [Accessed: Oct. 24, 2022].

[18] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Network and Distributed System Security Symposium*, San Diego, CA, 2014.

[19] G. working group. (2020, July 10). *Grant negotiation and authorization protocol*. [Online]. Available: https://datatracker.ietf.org/doc/charter-ietf-gnap/01/. [Accessed: Oct. 24, 2022].

[20] C. Lemmer-Webber, M. Sporny, M. S. Miller. (2020, Dec. 29). *Authorization capabilities for linked data v0.3*, World Wide Web Consortium Community Group. [Online]. Available: https://w3c-ccg.github.io/zcap-ld/. [Accessed: Oct. 24, 2022].

[21] D. Longley, M. Sporny. (2021, June 3). *Linked data proofs 1.0*, World Wide Web Consortium Community Group. [Online]. Available: https://w3c-ccg.github.io/ld-proofs/. [Accessed: Oct. 24, 2022].

[22] Agoric. (2021). *Endo* [Online]. Available: https://github.com/endojs/endo. [Accesed: Oct. 24, 2022].

[23] Agoric. (2021). *ECMAScript spec proposal for ShadowRealm API*. [Online]. Available: https://github.com/tc39/proposalshadowrealm. [Accessed: Oct. 24, 2022].

[24] C. Lemmer-Webber. (2021). *Spritely: Social worlds await.* [Online]. Available: https://spritelyproject.org. [Accessed: Oct. 24, 2022].

[25] C. Lemmer-Webber. (2021, July 18). *Spritely goblins v0.8 released!* [Online]. Available: https://spritelyproject.org/news/goblins0.8.html. [Accessed: Oct. 24, 2022].

[26] K. Varda. (2021). *Cap'n proto: introduction.* [Online]. Available: https://capnproto.org/. [Accessed: Oct. 24, 2022].

[27] Google. (2021). *Fuchsia*. [Online]. Available: https://fuchsia.dev. [Accessed: Oct. 24, 2022].

[28] J. A. Rees, "A security kernel based on the lambda-calculus," *Massachusetts Institute of Technology*, vol. 1564, 1995.

[29] M. S. Miller. (2011, Oct. 7). *Bringing object-orientation to security programming.* [Online]. Available: https://www.youtube.com/watch?v=oBqeDYETXME. [Accessed: Oct. 24, 2022].

[30] M. S. Miller, K.-P. Yee, J. Shapiro, *Capability myths demolished*, 2003.

[31] M. S. Miller, *Robust composition: Towards a unified approach to access control and concurrency control.* Baltimore, Maryland: Johns Hopkins University, 2006.

[32] M. S. Miller, C. Morningstar, B. Frantz, "Capability-based financial instruments," Proceedings of Financial Cryptography 2000, Anguila, BWI, 2000, pp. 349–378.

[33] J. Noble, S. Drossopoulou, M. S. Miller, T. Murray, A. Potanin, "Abstract data types in object-capability systems," IWACO, 2016.

[34] S. Peyton Jones, "A history of haskell: Being lazy with class," The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), 2007.

[35] M. Finifter, A. Mettler, N. Sastry, D. Wagner, "Verifiable functional purity in java," Proceedings of the 15th ACM Conference on Computer and Communications Security, 2008, pp. 161– 174.

[36] S. Wlaschin. (2015). *Using types as access tokens*, F# for Fun; Profit. [Online]. Available: https://fsharpforfunandprofit.com/posts/capability-based-security-3/. [Accessed: Oct. 24, 2022].

[37] E. Evans, *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley, 2004.

[38] C. Simpson. (2018). *Object capability discipline*, Monte Project. [Online]. Available: https://monte.readthedocs.io/en/latest/intro.html#object-capability-discipline. [Accessed: Oct. 24, 2022].

[39] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," Proceedings of the 16th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, ACM, 1989, pp. 60–76.

[40] T. Close, "Web-key: Mashing with permission," Proceedings of Web 2.0 Security and Privacy, 2008.

[41] N. Lœuillet. (2016). *Wallabag*. [Online]. Available: https://wallabag.it/en. [Accessed: Oct. 24, 2022].

[42] J. Schoning. (2019). *Espial.* [Online]. Available: https://github.com/jonschoning/espial. [Accessed: Oct. 24, 2022].

[43] S. SAS. (2005). *Symfony* [Online]. Available: https://symfony.com/. [Accessed: Oct. 24, 2022].

[44] T. Y. Team. (2012). *Yesod web framework.* [Online]. Available: https://www.yesodweb.com/. [Accessed: Oct. 24, 2022].

[45] T. P. Team (2017). *PureScript*. [Online]. Available: https://www.purescript.org/. [Accessed: Oct. 24, 2022].

[46] T. Bazaz, A. Khalique, "A review on single sign on enabling technologies and protocols," *International Journal of Computer Applications*, vol. 151, pp. 18–25, 2016.

[47] C. Michael, M. Gegick, S. Barnum. (2005, Sep. 12). *Complete mediation*, The Cybersecurity; Infrastructure Security Agency. [Online]. Available: https://us-cert.cisa.gov/bsi/articles/knowledge/principles/completemediation. [Accessed: Oct. 24, 2022].

[48] N. V. Limited. (2016). *Element*. [Online]. Available: https://www.element.io/ [Accessed: Oct. 24, 2022].