TU WIEN Informatics

Object Capabilities und deren Vorteile für die Sicherheit von Webapplikationen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Koppmann, BSc
Matrikelnummer 01125053

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar R. Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Wien, 12. Oktober 2021

Michael Koppmann — Edgar R. Weippl

Informatics

Object Capabilities and Their Benefits for Web Application Security

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Koppmann, BSc
Registration Number 01125053

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar R. Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Vienna, 12th October, 2021

Michael Koppmann	Edgar R. Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Michael Koppmann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Oktober 2021

Michael Koppmann

Danksagung

Zuallererst möchte ich mich bei Georg Merzdovnik bedanken, der mir die Chance gegeben hat, dieses Thema im Zuge meiner Diplomarbeit zu verfolgen.

Vielen Dank auch an die OCAP-Community, die seit Jahrzehnten an diesem Sicherheitsmodell arbeitet und es stetig verbessert.

Ebenso bedanke ich mich bei meinen Freunden, deren Input mir immer weitergeholfen hat und mit denen das Studienleben Spaß gemacht hat.

Schlussendlich möchte ich mich natürlich bei meiner Lebensgefährtin und meiner Familie bedanken, die mich stets unterstützt haben und für mich da waren.

Acknowledgements

First and foremost, I want to thank Georg Merzdovnik, who gave me the chance to work on this topic over the course of my diploma thesis.

Many thanks to the OCAP community, which continuously worked on and improved this security model over the last decades.

Likewise, I want to thank my friends, whose inputs always helped me and with whom life at university was fun.

Last but not least, I want of course to thank my significant other and my family, who always supported me and were there for me.

Kurzfassung

Heutzutage werden mehr und mehr Applikationen mit Webtechnologien wie HTML, CSS und JavaScript implementiert, die anschließend in Browsern ausgeführt werden. Das Web dient dabei als betriebssystemunabhängige Applikationsplattform. Dadurch ändert sich auch das zugrundeliegende Autorisierungsmodell, das nun nicht mehr ausschließlich von lokalen Betriebssystemkonten und Dateisystemberechtigungen abhängt. Stattdessen werden diese Konten nun von den Applikationen selbst implementiert, inklusive aller Absicherungsmaßnahmen und Sicherheitskontrollen, die dafür notwendig sind. Aufgrund der darin liegenden Komplexität führt dies jedoch dazu, dass Fehler in der Autorisierungslogik zu den häufigsten Sicherheitsschwachstellen bei Webapplikationen zählen. Die meisten Applikationen bauen auf dem Konzept der *Access Control List* auf, einem Sicherheitsmodell das bestimmt, wer auf welche Objekte zugreifen darf.

Diese Diplomarbeit präsentiert das alternative Autorisierungsmodell der *Object Capabilities* im Kontext von Webapplikationen, und wie es genutzt werden kann, um gewissen Schwachstellenkategorien vorzubeugen. Dafür wurde eine Fallstudie durchgeführt und ein Prototyp einer Webapplikation entwickelt, die auf diesem Modell aufbaut. Der Prototyp wurde daraufhin einer Sicherheitsüberprüfung unterzogen, wobei die Applikation auf die zehn häufigsten Webschwachstellen untersucht wurde. Ebenso wurde eine Modellevaluierung durchgeführt, bei der die grundlegenden Konzeptunterschiede beider Sicherheitsmodelle verglichen wurden. Dafür wurden Beispiele aus bestehenden Applikationen genommen, die auf Access Control Lists basieren. Die Ergebnisse der Analysen sind vielversprechend, zeigen jedoch auch auf, dass Erweiterungen in aktuellen Browsern notwendig sind, um Object Capabilities im Web noch sicherer zu gestalten.

Abstract

Nowadays, more and more applications are built with web technologies like HTML, CSS, and JavaScript, which are then executed in browsers. The web is utilized as an operating system independent application platform. As a consequence, the underlying authorization model changes, which now no longer depends only on operating system accounts and file system 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*, a security model which decides, who can access what object.

This diploma thesis presents the alternative authorization model of *Object Capabilities* in the context of web applications and how it can be used to prevent certain vulnerability classes. A case study was conducted for this and a prototype of a web application was developed that is based on this model. A security analysis was then performed on the prototype, where it was tested for the ten most common security vulnerabilities found in web applications. Afterwards, the model was evaluated by comparing the fundamental differences between these two concepts. Examples were taken from existing applications that are built upon access-control lists. The results of these analyses are promising, but they also show that extensions in current browsers are required to further improve the security for object capabilities on the web.

Contents

CHAPTER 1

Introduction

Exploiting security vulnerabilities for criminal activities has become a business which costs companies worldwide multiple billion U.S. dollars a year [1]. By 2026, the cybersecurity market size is forecast to grow to 345.4 billion U.S. dollars [2]. Issues in how the security model of modern web applications are designed are part of this problem, which this work tries to address.

The *Open Web Application Security Project* (OWASP) published their "Top Ten" document the first time in 2003 and it was updated regularly since then. The goal of this document is to raise awareness in the developer community about the most common security vulnerabilities in web applications and how to fix them. The latest version, as of September 10, 2021, is the OWASP Top Ten 2017[1] [3].

Injection Attacks are, according to the "Top Ten" document, the most common kind of attacks on web applications. An injection attack can be performed by crafting special user input, which the application does not recognize as data but as code which needs to be run. This can be, for example, a SQL injection, where the user gets access to the database, or an operating system command injection, where the user is able to perform system commands on the remote server. The common way to prevent these kind of attacks is to encode the user input data with respect to their current environment, such that the system does not interpret this data as code.

Broken Authentication is the second most common kind of security vulnerability, according to the OWASP. This refers to a broken authentication or session management system. This includes brute force attacks on the login, or using unexpired session tokens, which are still valid. Solving this problem includes many recommendations, such as implementing multi-factor authentication, enabling strict rate limiting for request, requiring complex passwords, and generating secure random session IDs are some of them.

[1]Since then the newer 2021 version was released.

Broken Access Control is the fifth most common security issue and describes that users can perform actions they should not be able to because the system is not properly enforcing all constraints. This allows an attacker to view sensible data or perform critical operations. Implementing allow- and deny-lists, role-based checks, monitoring, and rate limiting are common counter-measures.

The listed vulnerabilities and the underlying systems in the OWASP "Top Ten" document have some properties in common:

- The application has ambient authority and power, given by the operating system.
- Authentication and performing a requested action are two distinct steps.
- The security concept is built around the model of the *Access-Control List* (ACL).

By building the web application around the concept of capabilities, the *Principle of Least Privilege* (PoLP) [4] is built-in by design, reducing the risk of being vulnerable to 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 a group of simple techniques built around this concept, complete authorization frameworks can be implemented this way.

1.1 Aim of the Work

Authorization in software systems in the last few decades have been built on the concept of access-control lists. Many techniques and patterns were developed to mitigate risks and security problems that are inherent to this kind of authorization scheme. This thesis will present an alternative approach to authorization, based on *Object Capabilities*, that is not susceptible to attacks that are common in ACL-based systems.

The following research questions will be answered in this thesis:

1. Can vulnerabilities in authorization systems be prevented by design?
2. Is a capability-based system at least as secure as an ACL one?
3. Can the web be used as a platform for exchanging secure tokens?
4. How compatible is it with the rest of the ACL-based ecosystem?

The overall aim of this thesis is to produce a collection of security patterns which prevent certain classes of vulnerabilities by design, while focusing on maintaining the compatibility to the currently existing ecosystem of software products. These patterns should provide practical benefits and have to be usable in real-world applications.

1.2 Structure of the Work

The rest of the thesis is structured as follows:

- Chapter 2 describes the background behind the technologies and techniques that are going to be presented in this thesis.
- Chapter 3 shows the implementation of a software prototype that is built on top of the presented concepts.
- Chapter 4 presents the results of the security analysis that was conducted on the prototype.
- Chapter 5 evaluates the security aspects of the presented model compared to real-life software products.
- Chapter 6 gives an overview on related work and how the security of OCAP-based applications can be further strengthened.
- Chapter 7 concludes the findings of this thesis.

CHAPTER 2

Background

In this chapter we are going to explain concepts and terms that are used throughout this thesis as well as state-of-the-art practices. This overview, in addition to some historical background, serves the purpose to establish or refresh the necessary knowledge base to understand the ideas and results in the following chapters. Further explanations are given when needed.

2.1 Capability-based Security

What is a capability? 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. When we talk about capability-based security in this work, we mean systems that are built around this concept.

Capabilities are about *access control*. While the name implies that it is about what a system is capable of and *can* do, it is actually about what a system is *allowed* to do. Because the word “capability” is so overloaded with different meanings and because this concept overlaps with a lot of object-oriented programming principles, nowadays the term *Object Capabilities* (OCAP) is oftentimes 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 [6] and being the basis for sophisticated implementations of operating systems [7][8], hardware [5], kernels [9], file systems [10], and more. They are also not always used for security but safety. In the programming language *Pony*, which has a focus on safe, high-performance computations, capabilities are used to tag data to allow problem-free concurrency without locks [11].

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. For a real-world metaphor, a car key resembles a capability. It designates a resource, a particular car, and access rights, opening it, driving it, etc. Holding access to that key provides all the authority needed to use the car. This leads to one of the key properties of capabilities: the ability to *transfer* them. They allow to delegate authority. Giving that car key to someone else allows the other person to drive that particular car. In the digital realm, we can further limit the authority of that key by making it revocable, only valid between a certain time range, destroying it after the first use, or setting a hard limit on how many kilometers the car will drive.

The physical key can be replicated or forged, which does not have to be true for the digital one. *Unforgeability* is the second most important aspect of capabilities. It guarantees that they can only be accessed via certain ways: *creation*, *transfer*, and *endowment*. When someone creates a new object, they receive a capability for it with full access. Capabilities can be transferred so to get access to a capability someone else has to pass this capability. This also implies that, for example, when Alice wants to pass a capability for Carol to Bob she needs to possess references to both of them, which means that an object with no capabilities cannot leak data to somewhere else. Also, Carol and Bob cannot force the capability transfer as Alice must participate and decide if she wants to do that [12]. Objects can be given capability at creation, so an endowment is a creation combined with a transfer.

Capability-based systems follow the *Principle of Least Authority*[1] (POLA), as it follows from using object capabilities. If an object is only allowed to do few things it can also cause less damage. This allows for collaboration between untrusted parties, as the potential damage that can be caused by a malicious actor is reduced to a minimum.

2.1.1 Programming with Capabilities

A capability in the context of programming languages is still a reference to some piece of data. In object oriented languages, this is usually a reference to an object but it is not limited to that. A capability can also be a reference to a primitive type, a function, a closure, or other data types and, therefore, be also used with other programming paradigms like functional programming.

In order to guarantee unforgeability, the used programming language 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 give access to object references.

Transferability is given by the fact that data, or references, can be passed as arguments to functions. It turned out that OCAP security can be encoded as normal programming techniques by omitting global scope, passing arguments, and lexical scope [13].

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

Listing 2.1: “Counter” example in JavaScript showing OCAP programming

```
 1 function mkCounter() {
 2   let count = 0;
 3
 4   return Object.freeze({
 5     increment: function () {
 6       return count++;
 7     },
 8     decrement: function () {
 9       return count--;
10     }
11   });
12 }
13
14 counter = mkCounter();
15 entryGuard.giveCounter(counter.increment);
16 exitGuard.giveCounter(counter.decrement);
```

Listing 2.1 demonstrates a “counter” example in *JavaScript*. 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, as 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. Line 14–16 then shows how to use this as a security mechanism. The object 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. It is also an example for good software engineering in general. Protecting the internal state of the counter object is data encapsulation or data hiding. Passing the function as an argument is equivalent to *Dependency Injection* and fulfills two of the five principles of *SOLID*, an acronym coined by Robert C. Martin [14]; the *Interface Segregation Principle* and the *Dependency Inversion Principle*.

Similar to design patterns in object oriented design, several patterns emerged for programming with object capabilities. We will now present four of these patterns.

Revoker Listing 2.2 shows an example for the *Revoker* pattern. When Alice passes an object reference to Bob, she can never forcefully get that reference back. Early research work assumed that this limitation is a downside of object capabilities [15]. The Revoker pattern demonstrates how revocability can still be provided in an OCAP system.

Listing 2.2: Revocation example in JavaScript

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

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

The `mkRevocable` function takes a function as an argument and returns an object with two functions: `wrapper` and `revoke`. The `wrapper` can be passed to other objects, in this example to Bob, who can then proceed to interact with `foo`. If Alice later regrets that decision because Bob started to act strange, she can call the `revoke` function which she kept to herself. Then `revoke` will set the function in the closure to **`null`**, making all further requests to the wrapper by Bob unusable.

Membrane Listing 2.3 shows an example for the *Membrane* pattern. This can be used, for example, at the edges 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.

Listing 2.3: Membrane example in JavaScript

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

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.

Sealer Listing 2.3 shows an example for the *Sealer* pattern. This can be used to securely transfer data between multiple objects without revealing the content to everyone involved.

Listing 2.4: Sealer example in JavaScript

```javascript
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);
```

Similar to the counter example, the `mkSealer` function uses `sealed` as 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`. `seal` 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.

In the example, Alice passes a `sealer` to Bob. Bob can use this sealer to send Alice a secret. Bob does not have a reference to Alice but Carol has and Bob has a reference to her, so he sends Carol 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.

In a sense, this is an example for public/private key cryptography, protected by the runtime of the programming language, instead of real cryptographic primitives. The functions `seal` and `unseal` are the public and private key. The `data` argument is the plain text, while `box` can be seen as ciphertext.

Limited Use Listing 2.5 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 gets reduced by one each time the wrapped function is called.

Listing 2.5: “Limited Use” example in JavaScript

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

These patterns also compose very well together. For example, to create a one-time use, read-only, revocable file capability, these constructions only have to be stacked: `mkLimitedUse(1, mkReadOnlyFile(revokableFile))`.

In the end, it all boils down to two guiding principles that allow to reduce authority in program development and, furthermore, reduces the risk of intentional or accidental bad behavior:

1. No usage of global mutable state. This enforces that mutable data must be passed explicitly between objects, allowing to control 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.

User	/etc/passwd	/home/alice/secret.txt	/home/bob/shared.txt
Alice	(read)	(read, write)	(read)
Bob	(read)	()	(read, write)
Carol	(read)	()	()

Table 2.1: Access Control Matrix example

These principles are sometimes easier, sometimes harder to follow, depending on the programming language and tooling in use. For example, languages that do not support higher-order functions, functions which can accept other functions or have functions as return value, must fall back to alternatives like the *Command Pattern* [16].

2.2 Identity-based Security

An ACL is 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*. We will use the model as described by Lampson [17].

Table 2.1 shows an example with three objects and three subjects. User Alice is allowed to read the file `/etc/passwd`, read and modify `/home/alice/secret.txt`, and, because Bob gave her access to it, to read `/home/bob/shared.txt`. Bob is allowed to read `/etc/passwd` and to read and write to `/home/bob/shared.txt`. Carol has the least permissions and can only read `/etc/passwd`.

Multics was one of the first operating systems that featured access-control lists for file system permissions [4]. Current operating systems usually allow to set three classes of access: read, write, and execute. Some also use an additional *owner* attribute. In Linux, a file has an associated owner and a group and permissions can be set per owner, per group, and for all other users.

There are other identity-based authorization models, like *Role-based Access Control* [18] (RBAC) and *Attribute-based Access Control* [19] (ABAC). For our purposes their differences are not significant, so we will use ACLs as umbrella term for identity-based access control for the rest of this thesis.

These models have deficiencies when more than two subjects are involved. When passing through multiple subjects, information is missing that is required for correct access decisions. When Alice asks Bob to change a file for her at system C, she hands over a data string, for example, a file path. Bob faithfully complies and applies the requested change to the file at the given file path. System C, receiving a write request from Bob, verifies that he has the necessary authority to perform this action. In this scenario, Alice is able to name a file path that she has no access to but where Bob is allowed to apply changes. The ACL model is unable to authorize correctly in a multi-party system [20]. Exploiting this is also known as a *Confused Deputy* attack [21].

2.2.1 Comparison of ACL and OCAP

Miller et al. [15] described seven security properties to distinguish between the ACL and OCAP model with the purpose to clean up a lot of misconceptions which surrounded capability-based security for decades.

Property A. No Designation without Authority: All ACL models require some kind of shared namespace to designate resources so that subjects can name them. This reference must necessarily be independent from the representation of authority. Shared namespaces allow subjects in ACL systems to name arbitrary resources. This not only allows them to designate resources they have not been given access to by using known values but also to guess the names of resources. In a capability system, subjects hold references to objects. Each capability can consequently serve both as designation to the resource and as the set of permissions. This property allows capability systems to omit the need for a shared namespace.

Property B. Dynamic Subject Creation: Consequently, ACL systems also require a shared namespace in which to refer to subjects. The lists have to be continuously kept up-to-date about the set of subjects and resources. As subjects come and go, ACLs tend to prefer coarse-grained subjects like operating system accounts. This is not dictated by the ACL model itself but rather a consequence of applying a static model in a dynamic environment. In addition, usually only subjects with administrative privileges are allowed to create new subjects. In a capability system, capabilities are aggregated by subjects, which allow them to be defined on a fine granularity. Capability systems usually differentiate between single instances of a software component, like an object in a programming language or a running process of a program.

Property C. Subject-Aggregated Authority Management: ACL systems usually have an owner or edit property that allows subjects to edit the ACL for that particular resource. To restrict access to a resource, the subject needs the authority to change the whole ACL. Subjects in capability systems have their own capability lists and each subject can manipulate the authorities in their list.

Property D. No Ambient Authority: In ACL systems, a subject is usually not required to choose a specific authority. They only name a resource and the system handles the act of choosing the authority and performing the authorization. This is called *Ambient Authority*. With capabilities, by choosing to use a particular capability, the subject decides to exercise this authority. Thus, ambient authority is not required in a capability system.

Property E. Composability of Authorities: Subjects and resources are separated in an ACL system. Subjects cannot designate other subjects and cannot modify their authorities, as they are only aggregated by resources. In capability systems, subjects are also resources and resources are conceptually subjects. This allows to unify access and authorization and enables these systems to have deeply nested networks of relationships.

Property F. Access-Controlled Delegation Channels: Delegation in ACL systems requires a subject to have the authority to change the ACL of a resource. They cannot delegate authority directly to another subject. Having access to a subject or a resource by possessing the capability for it is a prerequisite in capability systems. When Alice wants to delegate authority to Bob, she has to possess a reference to him.

Property G. Dynamic Resource Creation: Unlike property B, ACL systems have no problem with dynamically adding new resources. The same is true for object capabilities but not all capability systems work like this. POSIX capabilities, for example, only provide a bounded set of flags which can not be extended or reduced.

Property	Model			
	ACLs as columns	POSIX capabilities	Unix file descriptors	Object capabilities
A: No Designation without Authority	○	○	●	●
B: Dynamic Subject Creation	○	●	●	●
C: Subject-Aggregated Authority Management	○	●	●	●
D: No Ambient Authority	○	○	●	●
E: Composability of Authorities	◐	○	◐	●
F: Access-Controlled Delegation Channels	◐	◐	◐	●
G: Dynamic Resource Creation	●	○	●	●

●=holds property; ◐=unspecified or workaround needed; ○=does not hold property

Table 2.2: Comparison of ACLs, POSIX capabilities, file descriptors, and OCAP with respect to the seven described security properties; based on Miller et al. [15].

Table 2.2 shows a comparison of different security models. POSIX capabilities and Unix file descriptors were added to this comparison because these systems are likely to be more familiar to the reader. Property E and F are not specified for the ACL model. POSIX capabilities are the only model missing property G because they only provide a fixed set of resources and only provide coarse-grained permissions. With Unix file descriptors property E can be implemented with pipes, while F holds but only by using ACLs. Object capabilities hold all described security properties.

Property B and G are required for the principle of least authority. We want to grant minimal subjects access to minimal resources. For this, we need to dynamically create new subjects and resources as required. Systems with property B but not E are not capable of revoking delegated rights, while systems with B but not F cannot confine delegated rights. Property D is crucial for preventing confused deputy attacks.

For a better understanding of how object capabilities would change the user experience in the context of desktop applications, assume the following situation: Alice wants to open a file in an editor. On current operating systems, the process looks as follows: Alice is logged-in and every action she triggers is performed with the authority of her user account. Applications that she starts run with the full authority she possesses. Alice opens the file manager, navigates to the file she wants open, and clicks on it. The default application, associated with the file extension, is automatically opened and the file path is passed to it. The application, running with the same authority as Alice, requests the operating system to open the file designated by the given file path. The request gets granted, and Alice can now edit the file in her editor.

This approach entails risks. Because the application is running under the same user account, it can do everything Alice is allowed to do. It can open other files in the background, search for secrets, encrypt all private data, or send copies of them over the internet. The application can do this because Alice can do this.

We now compare this to the capability-based approach. Alice opens her file manager. The file manager is part of a trusted computing base, which is given more authority than usual, and is able to display and open all files that Alice has access to. Keeping this trusted base small helps lowering the attack surface of the system. Alice clicks on the file she wants to edit and the associated application is started. Instead of passing the file path to the application, the file manager first opens the file itself and receives a file descriptor from the operating system. This file descriptor, a capability, is then passed to the application. With no additionally granted authorities, the application is confined and can only read and edit the single file that was passed in. It cannot cause other harm on the system. This also allows Alice to use features that are usually associated with increased risks, like running third-party macros which help her editing the file [12].

2.3 The E Programming Language

The *E* programming language was created by Electric Communities, originally as a set of extensions to the Java Virtual Machine [22][23]. It was used to create *WorldsAway*[2], an online graphical virtual world, based on LucasFilm's *Habitat*[3], which was one of the first graphical *Massively Multiplayer Online Role-playing Games* (MMORPG) [24].

E was open-sourced in 1997 and influenced later object capability languages like *Joe-E* [25], *Pony* [11], or *Monte* [26]. Unlike Java, it is an dynamically typed language and uses expressions over statements exclusively, which means that every function call returns a value. It was designed for secure distributed programming and is built for incorporating

[2]`https://www.youtube.com/watch?v=KNiePoNiyvE`, Accessed: 2021-09-09
[3]`https://www.youtube.com/watch?v=VVpulhO3jyc`, Accessed: 2021-09-09

the OCAP paradigms described earlier in this chapter. Variables in E are non-assignable, or mutable, by default and have to be explicitly declared as such with the keyword **`var`**. This is in contrast to languages like Java or JavaScript, where the opposite property holds, by declaring variables as **`final`** or **`const`** for them to be non-assignable. In our JavaScript examples given above, `Object.freeze()` had to be used so that the functions in the returned objects could not be modified later on. This is not necessary in the E language.

E, like Java, has safe pointers so object references are equal to object capabilities. But Java has mutable static variables which were removed in E. They allow that objects interact with other objects that are outside their reference graph, breaking the security model [27].

2.3.1 Darpa Browser

The Darpa Browser is the result of creating a capability-secure web browser that, even in the case of a maliciously acting render engine, would not cause any damage [28]. It is written in E and the system consists of three main components: CapDesk, PowerBox, and CapBrowser.

CapDesk is a secure, distributed file manager. It supports launching document-based applications and web browsers. CapDesk allows applications at installation time to negotiate endowments; authority that is automatically granted at launch time, like access to files that are required for running the application. It recognizes the act of opening a file as designating a file—the resource—as well as conveying the authority—the ability to open the resource.

PowerBox is the negotiator between applications and the user. It manages authorities endowed at installation time and can ask the user for more authority during the execution of a program. The PowerBox sits between trust boundaries and is written in a manageable size of code. This increased the confidence into the system, as the rest, although being larger in size, could not cause damage, as the necessary authority for this was collected in the audited PowerBox.

CapBrowser is a modular web browser that supports exchangeable HTML renderers. A malicious renderer was developed that tried to escape the browser's confinement to access user data. The project members could demonstrate that the malicious renderer could indeed not escape its sandbox, while the same renderer, running in an unconfined version of the CapBrowser, was capable of accessing valuable data.

The rest of the project focused on taming the Java APIs that were used for building this system. For example, calling `new File()` allows an application to escape the sandbox, so they had to made sure that such API calls were not possible.

2.3.2 Polaris

Polaris is a package for Microsoft's Windows XP that enforces the principle of least authority for applications running in it [29]. It is written in E and is an attempt to solve the problem with ambient authority, while still being able to run on a classic ACL-based operating system. Polaris includes many techniques previously used by CapDesk, like utilizing clicking on a file as an act of designation, installation endowments, and the PowerBox.

Existing applications can be "polarized", each instance being called a *pet*. Each pet is given a pet name, giving the user an opportunity to recognize individual instances, and can be given several permissions and file extension associations. In the background, a new operating system user is created per pet, providing the necessary isolation between them. A user could then, for example, create two pets of a web browser, one for accessing the internet and one for the intranet, while a third instance is only being able to access the local file system.

Visual cues, like the title or color of a window's title bar, are used to signal to the user that a specific pet is running. When the user opens a file in a polarized application, the file is copied to a folder where the associated user for that pet has access to it. A synchronizer manages the changes between the copy and the original file. If the application is malicious or has malware embedded, the only thing it can access is that single file, leaving the rest of the user's files safe.

The effectiveness of Polaris was limited by the interfaces provided by Windows XP. For example, any application could read every GUI event that was send to any application. The project never left the pre-alpha phase.

2.4 Capability Transport Protocol

A key property of capabilities is that they are *unforgeable*. Capabilities are enforced by a runtime and cannot be faked. If Bob wants access to Carol but Alice has the only capability to her, Alice has to explicitly pass that capability to Bob. In an object capability safe programming language, this can be achieved by simple argument passing, where the Alice object calls the public interface of Bob to pass a reference to the Carol object as an argument. OCAP languages do not allow pointer manipulation, so object references are unforgeable [27]. In the context of file management, the Alice process requests a file handle for the Carol file from the operating system and receives a file descriptor as integer. Although this descriptor is a plain integer, other processes cannot use it to get access to the file as the operating system manages a table per process where these descriptors are recorded. On Linux systems, however, a process can send a file descriptor to another, potentially lower privileged, process over a local `AF_UNIX` socket by using the `sendmsg` syscall[4]. These methods work only on a single machine or within a single runtime of a programming language though.

[4]`https://linux.die.net/man/7/unix`, Accessed: 2021-09-06

A *Capability Transport Protocol* (CapTP) allows to continue programming in an object capability oriented style but in a distributed environment. The programmer can continue working with objects and on the interactions between them, not having to care about them being local or on remote machines, while CapTP transparently manages the network interactions in the background. In comparison, usually applications are designed in a “local first, one process” style and are later heavily restructured to allow being run in a distributed setup. CapTP abstracts the latter away so programs can be written in the former, easier to reason about style [30].

In many OCAP languages and frameworks, like E, Goblin [31], or Monte, objects live in communicating event loops called *Vats*. Each system can host multiple vats, and each vat can host multiple objects. Objects can perform synchronous functional calls only within the same vat, but all objects, regardless of whether they are in the same or a remote vat, can communicate by asynchronous messages. These languages implement the actor model which provides a foundation for concurrent programming [32]. Each object, or actor, can send asynchronous messages to other actors as long as they have access to their addresses. Typical for actor model languages, messages are handled sequentially, providing temporal isolation from each other and avoiding certain problems associated with concurrent programming, like race conditions [33].

This process is also similar to how JavaScript’s event loop and promise functionality work. That is no coincidence, as many people who worked on E and its promises were and are participating in the development of modern JavaScript [34]. Erlang and Elixir can also be seen as object capability languages: they are type-safe languages, implement actors, process IDs are unforgeable capabilities, and each process has its own separate state as there is no global scope. Data can only be shared with explicit messages and actors can only send messages to process IDs they possess [27]. These messages can also be sent over the network to other Erlang or Elixir nodes, demonstrating some form of a capability transport protocol [35].

The rest of this section describes how CapTP works in E specifically.

2.4.1 Pluribus

Pluribus is the name of E’s capability transport protocol. It is used for communication between vats and allows to work over mutually suspicious networks. Trust cannot be guaranteed across server boundaries, so Pluribus is built with that assumption to work without full trust [27]. Assume that we have object Alice on `VatA` and object Bob on `VatB` and that Alice sends data to Bob. There is another object on `VatB`, Carol, who Bob trusts and with whom he shares data given by Alice. This is correct behavior and works as expected. Now assume that `VatB` is misbehaving, runs on untrusted hardware or software, or an unsafe language like *C++*. A malicious actor called Malice can now steal data from Bob. From the perspective of Alice, nothing has changed; the situation is the same as if Bob works with Malice voluntarily. If Alice does not want that, she never hands out that data in the first place.

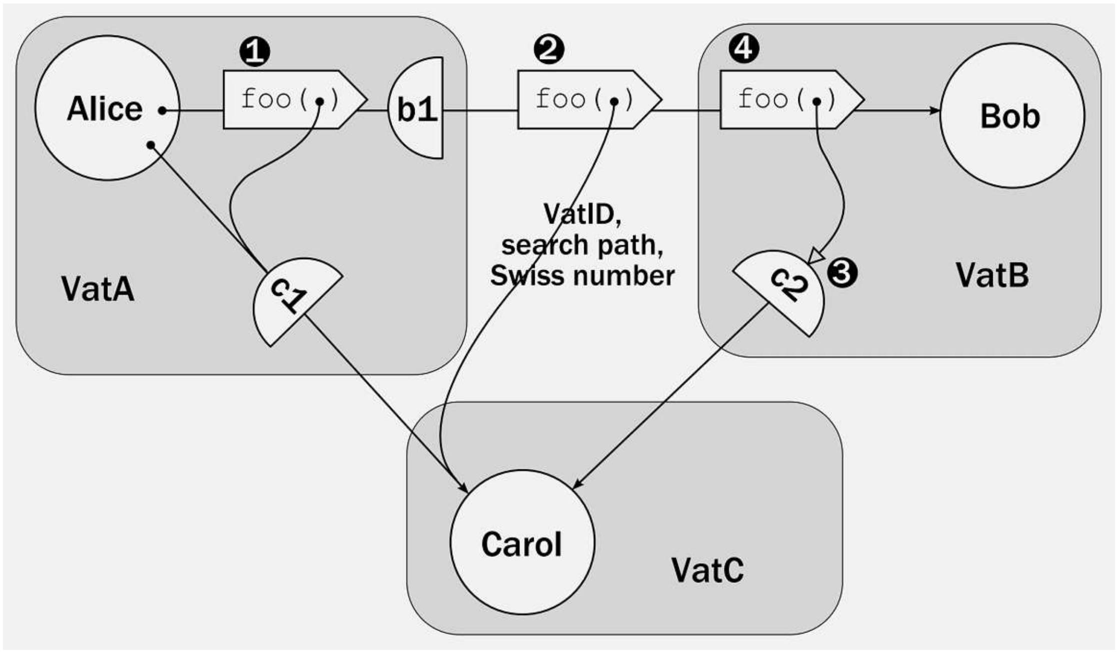

Figure 2.1: Distributed access control in E with Pluribus
Source [27]

Figure 2.1 shows a simplified version of how Pluribus works. There are three participating vats, each hosting one object. In this scenario, Alice in `VatA` wants to send Bob in `VatB` a message, containing a reference to Carol who lives in `VatC`. The half circles `b1`, `c1`, and `c2` are local proxies managing the communicating with the remote references. The message sent by Alice is a function call named `foo` and can be seen in several stages.

1. Alice calls `foo` on the local proxy `b1` in `VatA`. `c1` is passed as an argument to `foo`.
2. `b1` serializes the message and turns it into a network message for `VatB`.
3. `VatB` deserializes the message and performs a handshake with `VatC` to create the local proxy `c2`.
4. If the creation of `c2` was successful, `foo` is finally passed to Bob.

Sending capabilities over the network cannot give the same security guarantees that we have with a shared trusted platform like an operating system or a programming language runtime. CapTP in E works on top of *Transport Layer Security* (TLS) to get most of the benefits of safe pointers back. *Integrity* is guaranteed by secure cryptographic signatures. *Confidentiality* is guaranteed by encryption. *Unspoofability* is guaranteed by authentication, using public/private key pairs. *Unforgeability* cannot be guaranteed and is replaced by unguessable IDs which we will explain shortly.

When a vat is created, it generates a public/private key pair for authentication. The public key is its identity called *VatID*. When two vats perform a handshake, they verify that the other vat has the corresponding private key for its VatID. The *search path* is a list of IP addresses where the vat can be found. When an object is exposed externally for the first time, an unguessable cryptographically secure random number is generated; this is called a *Swiss number* and represents the identity of the object. In step 2, when `b1` serializes the function call with `c1` as an argument, the reference to Carol gets converted to a triple with her VatID, the search path, and her Swiss number. Objects in one vat can only communicate with objects in other vats when they know their Swiss number. The triple can also be serialized into a string and be communicated out-of-band over another secure channel. It can also be used for persistence and to bootstrap vats after a restart.

CapTP implementations like Pluribus also have two other useful features: *Promise Pipelines* and *Distributed Acyclic Garbage Collection* (DAGC).

Promise Pipelines: If multiple remote calls are required for some functionality, multiple round trips over the network are required for this to work. This adds latency to the system which at some point cannot further be reduced, as it is limited by the speed of light. Promise pipelines allow to reduce that latency. Promises are placeholders for values that are not yet present but which can already be used to specify further actions. This allows to send a sequence of actions to a remote target while reducing the amount of round trips to one with the final value [36].

Distributed Acyclic Garbage Collection: DAGC is cooperative garbage collection over several connected parties. It allows servers to communicate that it is safe to drop object references which are no longer required by any participating party. This helps reducing the required amount of storage for holding remote references [37].

2.5 Waterken: Web-Key

In the traditional web security model, resources are referenced by an URL. The request is then enhanced by the browser by adding a cookie which contains some kind of token. This token is used on the server-side to verify, if the request is authenticated or not. The token itself is a proof, given by the server to the client, that some kind of authentication ceremony, like a login, was performed. The authorization check on the server is then based on the identity of the caller.

This model has several drawbacks. As designation is separated from authority, every request is always performed automatically with the full authority of the user. This allows the possibility of confused deputy attacks, like *Cross-Site Request Forgery* (CSRF) [38] or *Clickjacking* [39]. Both attacks have in common that an attacker tries to trick the user's browser directly or indirectly to perform requests on their behalf. They only have to state an URL as the destination and the browser will provide the cookie for the user.

In a CSRF attack, an attacker lures an user to send requests to another server they did not intend to. Some action gets triggered when the user opens the web site, created by the attacker. This can be, for example, an HTTP POST request sent in the background that changes state on the target server. CSRF attacks work because the browser is implicitly attaching cookies to requests if it has some for the target domain. For this to work, the user has to be logged in at the target website. One way to prevent this attack is by requiring a secret token per user that needs to be attached to the request. This token can be included in the HTML form as a hidden field. An attacker would then have to know this secret value before they can prepare a malicious website. The *Same-Origin Policy* (SOP), a core security measure implemented in browsers, prevents them from reading any responses from other origins.

Classic clickjacking attacks work by designing the attacker's website in a way that the user thinks they click on buttons on this page but they actually click on a hidden embedded page behind it. This works for the same reason described for CSRF attack. The most common solution to prevent this attack is by setting HTTP headers which indicate that the target web page cannot be embedded in other origins. But this also prevents good behaving web pages from integrating these sites.

Waterken was an open-source web application server, developed at HP Labs, that implemented a concept called *Web-key* [40]. Waterken itself was at one stage implemented in *Joe-E* [25], a capability-safe subset of the Java programming language. Web-keys are a way to transport capabilities on the web by using URLs with access tokens. Designation and authority are combined in a single URL, therefore, eliminating the risks of confused deputy attacks. As an attacker would need to know the appropriate access token to perform the attack, they could, at this stage, perform the request themselves.

Web-keys are a way to transport capabilities over HTTPS, but as this is not a capability transport protocol it does not provide all the necessary security requirements and has several drawbacks. Mainly, the property of unforgeability is missing and it is replaced with *unguessability*. We provide a comparison of different methods to transmit data over HTTP in 4.4 Protection Level of Web-keys while new proposals are shown in 6.1 New URI Schemes.

When web-keys were originally proposed, they used cryptographically secure random generated bit strings with a length of 64 characters. They were seen as having enough entropy to protect against brute force attacks, while keeping them short enough to allow transcription of URLs. To transmit them over HTTPS, they were Base32-encoded resulting in a final length of 13 characters.

These tokens are then transmitted as part of the URI *fragment*. The fragment was chosen to avoid leakage of the tokens in the HTTP `referer` [*sic*] header. The referrer header gets attached to a request by a web client when browsing to another web page, containing the link where the hyperlink was displayed. Thus, when linking to a third-party web-page, the referrer would then include the access token in the URL. RFC 2616, which specifies HTTP 1.1, states that clients must not include the fragment in the referrer header [41].

But the fragment is also not send to the server when opening the link. To restore functionality, JavaScript is used to extract the token from the fragment. A second request is then made by JavaScript in the background attaching the token as part of the query string which the server then can parse. Creating a request like this does not result in the creation of the referrer header. Modern web applications have more control over this default behavior of browsers by setting the HTTP `Referrer-Policy` [42]. This policy can be used to completely disable the creation of the referrer header or to only send it for same-origin requests. (and several other constellations) Thus, by utilizing the correct referrer policy the requirement on JavaScript can be lifted again, as using the query string directly in an URL is not prone to that leakage anymore.

2.5.1 Deep Linking

Web-keys, in contrast to cookie-based authentication, can be used to create privileged URLs. Different web applications can work together by freely embedding pages with web-keys and linking them, without the restrictions of the same-origin policy.

Compared to the solution for cross-site request forgery attacks, where the security measure against them relies on the protection of the same-origin policy (the hidden token is not readable by other origins), web-keys do not require it as the URL itself is the unguessable secret.

By representing capabilities as URLs, users are allowed to use tools they already know to manage them, for example, the bookmark manager integrated in the browser. As users can store authorized links, they are not required to come up with their own passwords and secrets, reducing the need for login pages. Web applications that can be used without a login are resistant to phishing attacks.

2.6 OAuth 2.0

The *Open Authorization* protocol version 2.0 (OAuth 2.0) [43] is one of the most popular protocols to implement *Single Sign-On* (SSO) [44]. It is the successor to OAuth 1.0 and was released in October 2012 as RFC 6749. This protocol allows the owner of a resource to give third-party applications limited access to an HTTP web service, where resource data are provided. During the process, the resource owner obtains access tokens from the authorization service which can then be passed over to the third-party application which in turn then uses these tokens to access the protected resources hosted on the resource server. These tokens can be limited in scope such that authority is reduced to a minimum and can be revoked. This allows the delegation of authority without the need to provide full credentials like username and password to potentially untrusted parties.

Together with OAuth 2.0, RFC 6750 [45] with the title “The OAuth 2.0 Authorization Framework: Bearer Token Usage” was released. This document describes how the access tokens mentioned above should be transferred to prevent misuse and to protect them, for example, by making TLS mandatory. There are three ways to send authenticated requests in OAuth:

1. Using the `Authorization` HTTP request header, prepended by the string `Bearer`. This can look like this: `Authorization: Bearer abc-123`.
2. As part of a form-encoded HTML body parameter. The parameter is named `access_token`. It is recommended to prefer method 1, if possible.
3. As HTTP URI query parameter called `access_token`. This method is discouraged because of the security issues associated with it. See 4.4 Protection Level of Web-keys for our comparison.

The key difference between an OAuth bearer token and a capability is that bearer tokens are not bound to a reference to the resource and are, thus, still susceptible to confused deputy attacks. Some suggested solutions for that problem can be seen in Section 6.1 New URI Schemes. Both techniques are examples for explicit authorization compared to implicit authorization by web cookies.

2.6.1 Authorization Code with PKCE Flow

OAuth 2 provides several flows, each specifying how to request access tokens, adapted for specific use cases. Most of them are now considered to be obsolete, as newer, more secure flows can replace them. *Authorization Code with PKCE* is currently one of the secure flows according to the OAuth standard and is, furthermore, the recommended flow for the upcoming OAuth 2.1 standard [46].

This flow has three interacting parties: A resource owner, a client app, and an authorization server. Using the access token for fetching resources involves other parties. The resource owner is usually represented by an user agent, a web browser, performing the actions for the user. It accesses the client app which in turn first generates a random secret. The hash of this secret is then returned to the browser using a cryptographically secure algorithm, in this case SHA-256. The browser then redirects to the authorization server with the hash which gets stored. The user submits their credentials for the login and upon success the authorization server redirects the browser back to the app with an authorization code. The app now has everything to request an access token. It contacts the authorization server with its client ID, the original secret, and the authorization code. The authorization server verifies the validity of the secret by performing the same hash algorithm and if client ID, hash, and code match it finally returns the access token to the app. Figure 2.2 presents a graphical representation of the flow.

In this flow, several redirects are performed, implying that secrets are transmitted in the query string so a similar security level to web-keys should be expected. But several mitigations are in place to reduce the attack surface. The generated secret, its hash, and the authorization code are one-time use and have to be regenerated for each new

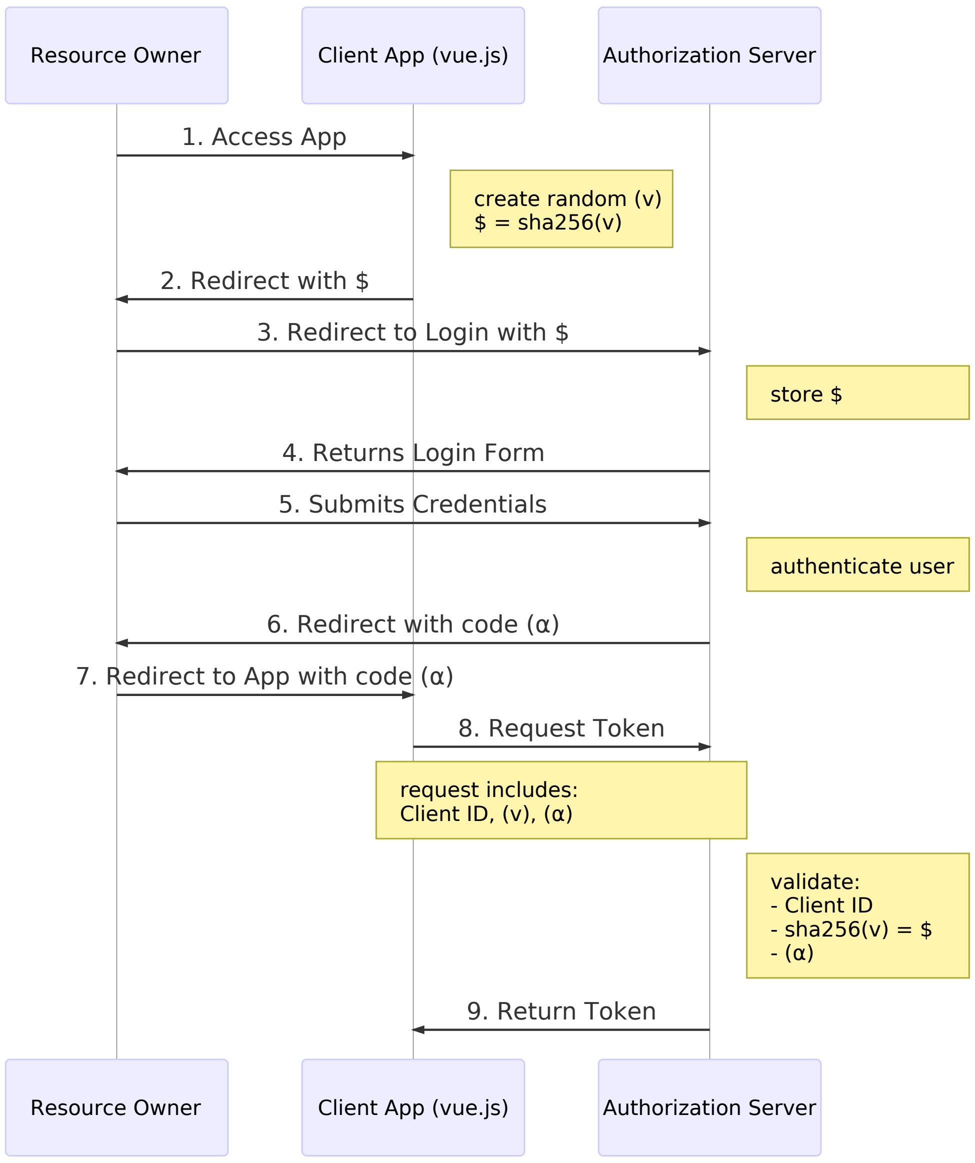

Figure 2.2: OAuth authorization code with PKCE flow.
`https://developer.okta.com/blog/2019/08/22/okta-authjs-pkce`

authorization request. In addition, the code is also short-lived to reduce the open time window for attacks. Access tokens themselves are usually also short-lived. When they expire, either a new authorization flow is started, or refresh tokens are used. Each time an access tokens gets returned a refresh token is returned too. They are long-lived and are also only valid for a single use. When the access token expires, the refresh token can be used to get a new pair of tokens, thus, not requiring the user to login again.

2.6.2 Rich Authorization Requests

Although OAuth provides scopes to limit the authority of access tokens, they are still very static and can only be used for coarse-grained permissions. *Rich Authorization Requests* [47] (RAR) is an extension that allows for fine-grained authorization grants. This allows for scenarios like: "Give me a token that let me send *123* Euro to the account with the IBAN *abc*." Requesting such tokens can be done by using the new `authorization_details` request parameter. It contains serialized JSON with one required field: `type`, which specifies the authorization data as a string and which determines the other allowed fields in the JSON.

Listing 2.6: Rich Authorization Request example

```
[
  {
    "type": "payment_initiation",
    "actions": [
      "initiate",
      "status",
      "cancel"
    ],
    "locations": [
      "https://example.com/payments"
    ],
    "instructedAmount": {
      "currency": "EUR",
      "amount": "123.00"
    },
    "creditorName": "Merchant123",
    "creditorAccount": {
      "iban": "abc"
    }
  }
]
```

The draft also defines a set of common fields that may be part of the JSON, depending on the type, but additional fields can also be used:

- `locations`: Strings representing the locations of resources or resource servers; typically URIs
- `actions`: Strings representing which kind of actions to be performed on the resource
- `datatypes`: Strings representing which data types are requested from the resource

- `identifier`: ID of a resource
- `privileges`: Strings representing the levels of privilege that are requested

Listing 2.6 shows the request for a token that fulfills the scenario described above. These fine-grained access tokens, in combination with 6.1 New URI Schemes, could lead to a secure exchange of capabilities on the web.

CHAPTER 3

Eselsohr — A Case Study on Capability-based Security

This chapter describes the implementation of a web application prototype, which capability-based techniques were used, and which benefits they bring in. This demonstrates how the concepts described in the previous chapter can be used in practice to improve the overall quality of application code.

Eselsohr is a web application created during the course of this thesis to gain a better understanding of the practical applications of capabilities-based techniques. It will also be used for the model evaluation and will be compared against ACL-based projects with similar features. The prototype implements a simple bookmark manager where URLs can be stored in collections, which can 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.

Data Can Be Shared Between Instances: Eselsohr implements file-based persistence. The same URL can be used on multiple instances as long as all those instances have access to the referenced file.

Shareable Permissions: Access to a collection is granted with URLs. New URLs with reduced permission sets can be created by people. Links can also 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.

Scalable: Eselsohr uses concurrent and parallel programming techniques on a per-collection basis. By passing in capabilities, which allows for side-effects, managing concurrent code is made easier compared to traditional approaches with shared mutable state, locks, and mutexes.

Easy Deployment: Eselsohr is distributed as a statically-linked binary with no external runtime dependencies. This makes this service comparatively easy compared to other, more complex setup guides. During development, this was a key factor and several design choices are based on it, for example, the reduced use of cryptography to avoid the need for key rotation.

The following sections describe how a capability-based approach is used in different layers of the application to provide features, improve maintainability, or security. The goal is to reduce the ambient authority and stick to the principle of least authority.

3.1 Purely Functional Programming in Haskell

Haskell is a statically typed, immutable by default, purely functional programming language with lazy evaluation [48].

- Statically typed means that the type checking, that proofs the soundness of the program, happens at compile time.
- Immutability means that once a value has been created it cannot be modified afterwards. A change to a value is equivalent with creating a copy of it with the new value applied.
- Functional purity means that a function always provides the same output for the same input. This implies that functions only work with their input arguments, do not depend on global state, and do not perform any side-effects, like printing to a console, writing to a file, sending a request to the internet, etc.
- Lazy evaluations means that the evaluation of an expression is delayed as far as possible. This is made easy with pure functions, as their result do not depend on the evaluation order.

One unusual aspect of Haskell is its explicitness on side-effects. All side-effects must happen in the **IO** data type.

Listing 3.1: “Hello, World!” example in Haskell.

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

Listing 3.1 shows a simple “Hello, World!” program in Haskell. The function **putStrLn** prints the given string to `stdout`. As printing to the console is a side-effect, the return type of this function must be **IO**, everything else will result in a compilation error. Once in **IO**, all side-effects are permitted.

Another restriction is that impure functions can call pure ones but not vice versa. The **IO** type is tainting and has fewer benefits than pure functions have, like composability, determinism, and the property to test them in isolation. Therefore, Haskell programmers tend to design their applications in a way that puts their pure logic at the core of the application, while interacting with the outside world via a thin layer of impure functions. This architecture pattern is also known as “Functional Core, Imperative Shell” [49].

Although Haskell is a functional programming language and does not have a concept of an object, many object capability patterns can still be applied with some modifications. As object capability languages also try to achieve functional purity [50], Haskell’s strictness on the separation between values and effects fulfills this goal.

Eselsohr uses types in two ways as a kind of capability to achieve the principle of least authority: as access tokens within the runtime and to limit the possible effects it can have.

3.1.1 Types as Capabilities

As described in 2.6 OAuth 2.0 external authorization systems work like this:

1. A client wants to access a resource. They must proof that they are authorized to do this.
2. The client presents claims, like their identity and the requested scope, to an authorization service.
3. This service performs the necessary authorization checks and returns a cryptographically signed token to the client.
4. 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 behavior without using any cryptography but are secured by the runtime of the language [51]. This can be achieved by using types with private constructors, which are functions that can create values of that specific type but which are not exported outside of their respective module. Other modules, therefore, can not 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.

A privileged function, such as a database accessing one, would then not require 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 3.2: Example authorization module

```haskell
module Authorization
    ( AccessToken
    , AccessArticle(..)
    , DeleteArticle(..)
    , getData
    , 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 -}
```

```haskell
        then Just (AccessToken (AccessArticle articleId))
        else Nothing

-- | 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
```

Listing 3.2 shows such an authorization module which handles access tokens. The type `AccessToken` wraps a generic type `a`. Its 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 an 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.

The `Maybe` type represents potential failure and states that this function can return a value or not, depending on the given argument. The caller then is forced by the compiler to handle both possible outcomes. This is a type-safe alternative to `null`, known in languages like Java, C, C#, Python, Ruby, JavaScript, and more, where a possible return value of `null` is not explicitly stated in the return type of the function. Dereferencing a null reference results in runtime errors and Tony Hoare called this his “Billion Dollar Mistake” [52].

Listing 3.3: Example database module

```haskell
module Database where

{- import required types and functions -}

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

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

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

Listing 3.3 shows how a module with privileged functions can use these access tokens to guarantee that the caller performed an authorization check.

The function `getArticle` is expecting a value of type `AccessToken AccessArticle` instead of a plain `Id`. Therefore, it is not possible to call this function without calling `accessArticleToken` before. The token gets unwrapped with the `getData` function and the `articleId` is extracted via pattern matching. Finally, the `getArticleFromDB` function is called with the `articleId`, resulting in the requested effect of fetching the article. The function `updateArticle` works the same but in addition expects a new article argument.

A function which requires a different kind of authority is `deleteArticle`. It does not work with the `AccessToken AccessArticle` privilege but expects a value of type `AccessToken DeleteArticle`. The corresponding `deleteArticleToken` function also performs different authorization checks.

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 3.4: Function for creating access tokens without authorization

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

Listing 3.4 shows a function which takes an argument with a generic type `a` and puts it in the `AccessToken` type. The prefix `unauthorized` serves as a hint and can be detected during code reviews or by automatic tooling. Alternatively, unauthorized versions of the database functions can be provided.

This technique can not only 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, like 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.

The *sqlite-simple*[1] Haskell package, a library for using a SQLite database, utilizes a `Query` type, which is used by the functions that execute SQL code. The `Query` type is not compatible with regular strings and can not be easily mixed with external user input. Therefore, proper parameter substitutions must be used.

The Haskell package *lucid*[2] provides a domain specific language to generate HTML. Again, functions from the *lucid* package do not work with plain strings and external input has to be transformed into HTML by calling the `toHtml` function. This guarantees that output encoding has been performed.

Using types that enforce invariants and that represent concepts in the domain of the application is a pattern also known as “value object” in the realm of domain-driven design [53].

3.1.2 Types for Explicit Side-effects

The `IO` type gives us too much ambient authority. To achieve POLA, we want to reduce the possible effects to a minimum. Object capability languages, like Monte, only allow the import of IO-providing functions at the entry point of a module [26]. These functions are then passed along as arguments until they get called. This reduces the amount 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 [54].

To understand how a purely functional programming language can achieve any side-effects at all, we introduce the concepts of *Monads*. Monads, originally coming from category theory [55], are a generalization over sequential control flow. Two functions are required for a data type to have a monadic interface: one for wrapping any value within the monad and one for composing functions that output monadic values.

The former is called `return` in Haskell, while the latter is called `bind` but most of the time the infix operator >>= is used. To avoid confusion with the `return` statement in imperative languages, the equivalent `pure` function is used. The implementation also has to follow the monad laws [56], which are out of scope for this thesis.

To demonstrate how monads can be used to describe sequential execution steps, we will use `Maybe` again, which also implements a monadic interface.

Listing 3.5: Functions that return a monadic value

```haskell
validateInput     :: String -> Maybe String
canonicalizeEmail :: String -> Maybe Email
```

[1]https://hackage.haskell.org/package/sqlite-simple, Accessed: 2021-09-25

[2]https://hackage.haskell.org/package/lucid, Accessed: 2021-09-25

```
lookupInMemory    :: Email  -> Maybe User
```

Listing 3.5 shows three function signatures. All three functions expect a plain value as input and return a value wrapped in a `Maybe`, as all of those functions can fail depending on the input.

Listing 3.6: Function that calls other functions returning Maybe in sequence without using monads

```
lookupUser :: String -> Maybe User
lookupUser input =
  case validateInput input of
    Nothing     -> Nothing
    Just vInput -> case canonicalizeEmail vInput of
      Nothing    -> Nothing
      Just email -> case lookupInMemory email of
        Nothing     -> Nothing
        Just user   -> pure user
```

Listing 3.6 shows the function `lookupUser` which takes a `String` as an input and returns a `Maybe User`. This function does not use the monadic interface of `Maybe`, so after each function call the `Maybe` value has to be unwrapped to extract the plain value contained within. This code is similar to imperative code, where checks are required to avoid the dereferencing of `null`.

Listing 3.7: Monad implementation of the Maybe type

```
instance Monad Maybe where
  return x        = Just x
  (Just x) >>= f  = f x
  Nothing  >>= _  = Nothing
```

Listing 3.7 shows an implementation of the monad interface for the `Maybe` type. When the function `return` gets called with a value `x` it gets wrapped in a `Just`. When `bind` is called with a function `f` and a `Just` value `x`, then the value is applied as an argument to the function. If the given value is `Nothing`, then `Nothing` will be returned, therefore, short-circuiting the remaining function executions. With this we can now rewrite the example from listing 3.6 as follows:

Listing 3.8: Function that calls other functions returning Maybe in sequence using monads

```haskell
lookupUser :: String -> Maybe User
lookupUser input =
  validateInput input
    >>= \vInput -> canonicalizeEmail vInput
    >>= \email  -> lookupInMemory email
    >>= \user   -> pure user
```

Listing 3.8 shows how monads can be used to compose functions that return a monadic value. The values on the right hand side of the bind operators are lambda expressions, binding the result of the previous function to a new name, which can be used as an input for other functions. If any function in that chain returns `Nothing`, then the whole function returns `Nothing`.

Haskell provides syntactic sugar, the do-notation, which allows writing monadic functions in an imperative-looking way. The code in listing 3.9 desugars to the code seen in listing 3.8.

Listing 3.9: Function that calls other functions returning Maybe in sequence using do-notation

```haskell
lookupUser :: String -> Maybe User
lookupUser input = do
  vInput <- validateInput input
  email  <- canonicalizeEmail vInput
  user   <- lookupInMemory email
  pure user
```

The `IO` data type uses the same mechanism to represent this "do this, and then this, and then this" semantics, but for side-effects. The Haskell program itself can stay functionally pure as only the sequential steps are described but not executed. The Haskell runtime, which executes the program, is impure and exchanges data with the outside world. `IO`'s data constructors are not exposed outside of its module so values wrapped in `IO` cannot be extracted[3]. This is why impure function calls are tainting.

Haskell's mechanism to generalize behavior and patterns, like monads, over multiple data types is called a type class. Examples of other type classes are `Eq`, for checking

[3]Escape hatches like `unsafePerformIO` are ignored for simplicity. They are required for some use cases but have to be used with care, hence the prefix `unsafe`.

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 3.10: Type class of Eq

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

Listing 3.10 shows the type class of **Eq**, which includes the equal (==) and not equal (/=) functions.

Listing 3.11: Type signature of a polymorphic function with Eq constraint

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

Listing 3.11 shows the type signature of a polymorphic function called **uniq** which, similar to 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 amount of possible side-effects in our program and to simulate the approach taken by object capability languages, Eselsohr uses a custom monadic data type and type classes to explicitly declare all possible side-effects per function.

Functions that work with side-effects use, instead of returning in **IO** directly, a generic type **m**. This type is then constrained by type classes that represent effects. The business logic of the application, thus, stays 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 the **App** monad in Eselsohr.

Listing 3.12: Example function showing effect type classes for a custom monad

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

Listing 3.12 shows such a function. Here, **createArticle** takes an **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`.

Listing 3.13: Example effect class representing the access to time

```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 3.13 shows the implementation of the `MonadTime` class for the `App` monad, which uses an `IO` function from the *time*[4] 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 gets 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 [54], we simulate the behavior of object capability languages like Monte, where passing side-effecting functions happens explicitly. This is also a form of the *Sealer* pattern (see 2.1.1), where only functions that were given a specific type constraint can access the `IO` functions contained in the `App` monad.

We conclude that types can be used, similar to capabilities, to improve the security of a program. The value of the type is the object reference, while the set of functions that can work with this type are the access rights.

3.2 URLs as Capabilities

Eselsohr uses web-keys (see 2.5 WATERKEN: WEB-KEY) to transmit access tokens over HTTP. This was a compromise in the design decision. As Eselsohr is a web application that does not use JavaScript, the available options for transferring data are limited. For a comparison of the several methods, see 4.4 PROTECTION LEVEL OF WEB-KEYS. Web-keys were chosen as they are compatible with a JavaScript-free web application while still providing an usable user experience.

[4]`https://hackage.haskell.org/package/time-1.12/docs/Data-Time-Clock.html#v:getCurrentTime`, Accessed: 2021-09-25

Listing 3.14: Example URL to access a single article resource

```
https://eselsohr.local/articles/example-title?acc=
QMANQJKQCLCDXJT5DITHDRPVMFQF4MN3MCKUBFZVIPHU52S72SPX2CI2GU
```

Listing 3.14 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 UUID 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 binary 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 that these permissions have to be checked before the resource can be loaded. (see 3.1.1 Types as Capabilities)

Users can create new URLs for each page with restricted permissions and expiration dates, implemented with the *Membrane* pattern[5]. (see 2.1.1) This makes it possible to create, for example, read-only or append-only access to certain resources. Working with fine-grained permissions allows for dynamic use-cases which 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 (see 2.1.1).

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

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

3.3 Reducing Ambient Authority

The Eselsohr project provides hardening recommendations that reduce the attack surface of the service. As all common desktop operating systems are based on user-based access control, this approach does not use object capabilities. It is still a valuable addition to

[5]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 overall security of the program and shows that the different security models can be combined together in existing systems.

By minimizing the ambient authority that are available to the running process, exploitation is made harder, as the attacker is left with less available tooling on the system. As Haskell allows to build statically-linked binaries, only a minimal set of files is required at runtime.

Eselsohr currently provides two approaches that utilize the namespacing features offered by the Linux kernel to isolate the process from others:

Hardened systemd Service: When running a program as a service by using a systemd unit file, several sandboxing options are available. Access to the file system, to the network, and to kernel APIs can all be limited or completely disabled. This is done by creating dynamic users per service run, separate namespaces for the file system, firewall rules, and filtering syscalls. Exploiting the service would gain the attacker only access to an almost empty system. By only allowing the minimum set of features that are required to run Eselsohr, the principle of least authority can be uphold.

Minimal Docker Image: Docker images usually use well known Linux distributions like Debian, Ubuntu, or Alpine as base images and add their packages on top of them. This results in images that include a large set of packages found in a typical Linux installation, each one potentially vulnerable or usable by an attacker. So called *Distroless*[6] Docker images reduce the amount of files to a minimum, only containing what is necessary to run the application. Statically-linked binaries can even use the smallest available image, only containing certificates for trusted TLS connection, a `/etc/passwd` entry for an user, a `/tmp` directory and timezone data. It does not come with a shell, no glibc, or any other common package.

6 `https://github.com/GoogleContainerTools/distroless`, Accessed: 2021-08-24

CHAPTER 4

Security Analysis of Eselsohr

This chapter describes the security analysis of the implemented prototype, which vulnerability classes can be mitigated by capability-based techniques, and where new challenges arise.

4.1 Threat Model

As prerequisite for the evaluation of Eselsohr, a threat model was required. Given the recommendation that this service is primarily meant to be self-hosted by the users we excluded malicious administrators. Furthermore, as the currently stored data was not classified as sensitive information, and to keep the deployment simple, more advanced protection measures like encryption at rest by the application itself were not deemed necessary. This leaves us with the following list of attackers:

Remote network attacker is an attacker trying to exploit the application, either by performing social engineering attacks on the victim, or by attacking the public functionality exposed by the application.

Local network attacker is an attacker controlling the local network, being in a man-in-the-middle position, or controlling the infrastructure where the application is running on.

Local physical attacker is an attacker, who either has access to the victim's device, or is in physical proximity to observe the victim using the application.

4.2 Top 10 Web Application Security Risks

The *Open Web Application Security Project* published their “Top Ten” document the first time in 2003 and it was updated regularly since then. The goal of this document is to raise awareness in the developer community about the most common security vulnerabilities in web applications and how to fix them. The latest version, as of September 10, 2021, is the OWASP Top Ten 2017 [3], which covers these categories:

A1:2017-Injection: Untrusted user data gets interpreted as code, giving the attacker (partial) control over the server.

A2:2017-Broken Authentication: Incorrectly implemented authentication and/or session management, allowing the attacker to exploit identity-based security measures.

A3:2017-Sensitive Data Exposure: Sensitive data are not properly protected and can leak without elaborate attacks.

A4:2017-XML External Entities (XXE): Misconfigured XML processors evaluate external entity references in a way such that they can disclose internal files or be used for remote code execution.

A5:2017-Broken Access Control: Access to functions or data is not properly protected and can be used by unauthorized users.

A6:2017-Security Misconfiguration: As software commonly comes with insecure default configurations, the whole application stack needs to be reconfigured securely, while updates and patches must be applied regularly.

A7:2017-Cross-Site Scripting (XSS): Similar to “A1:2017-Injection”, user input gets interpreted as JavaScript code or HTML, but the attack is executed in other users’ browsers, not the server.

A8:2017-Insecure Deserialization: Untrusted input from attackers can cause malicious side-effects like remote code execution, injection attacks, or privilege escalation attacks.

A9:2017-Using Components with Known Vulnerabilities: A vulnerability in a third-party component gives an attacker the same power as a vulnerability in the application itself because they all run with the same privileges.

A10:2017-Insufficient Logging & Monitoring: A lack of logging or monitoring allows attackers to stay undetected for a long period of time.

Capability-based security does not help with all these problems, so the categories "A4:2017-XML External Entities", "A6:2017-Security Misconfiguration", and "A10:2017-Insufficient Logging & Monitoring" will not be further discussed in this thesis.

After this security analysis was conducted, the OWASP published the newer 2021 version of their Top Ten document[1]. However, the changes do not invalidate the findings of this thesis. Specifically authorization problems are now ranked even higher than before. This only highlights the need to rethink authorization in web applications.

4.3 Evaluation Results

A security analysis on the prototype was conducted, where the application was tested for the vulnerabilities listed above. The *OWASP Web Security Testing Guide*[2] (WSTG) was used as a guiding document. This covers the most common attacks on web applications. Protections against those provide a good security base line.

4.3.1 Injection

Any external source of data can be seen as untrusted, as there are no guarantees what kind of data will be provided. Depending on its usage, untrusted data may be interpreted as code, resulting in the loss of data, denial of service, or remote code execution. Therefore, it has to be encoded for the target environment, such that it is only handled as data, not code.

Eselsohr processes user provided input at several places. Environmental variables at startup, serialized data for authorization and persistence, as well as user input for article URIs and titles. Only the latter two are passed to an interpreter, an HTML renderer, but this falls into the category of XSS, which we will discuss later.

Assuming that the project would, for example, utilize a SQL database for persistence, two mitigating factors come into play: The type-driven approach to application design would make a string concatenation of a SQL query and user input a compile time error. (see 3.1.1 Types as Capabilities) In addition, because collections are stored in separate files, an attacker could only attack their own collection as they do not know access tokens for others.

4.3.2 Broken Authentication

Common web applications base their authorization decisions on the identity or the group association of the user. During a login sequence, the user is authenticated and a proof for

[1] `https://owasp.org/Top10/`, Accessed: 2021-10-11

[2] `https://owasp.org/www-project-web-security-testing-guide/v42/`, Accessed: 2021-08-27

this, usually in the form of a randomly generated token, is used by the client for the rest of the session's lifetime. As this token gives full control over the session of the user, it is important to protect it during its use and to invalidate it as soon as it is not required anymore.

The login process also requires additional protection. The most common way of authentication is providing a username and a password. Attackers have access to millions of valid credentials, collected results of hundreds of data leaks, which they can use to try logging in on any web page—this attack is also known as *Credential Stuffing* [57].

Capability-based security does not depend on authentication for authorization decisions, and our prototype does indeed not implement authentication, so it would be easy to say that this category also does not apply. There is no login page, so we do not have to protect it. No user provided password or suchlike are processed, thus, there is no risk of credential stuffing, and no further measures, like checking for weak passwords, are necessary. Instead, only high entropy, server-side generated secrets are in use, so brute force protection is not required.

We still rely on tokens that are passed between server and client for authorization. They are either transmitted in the URL or inside the HTTP body. For a comparison how well they are protected, see 4.4 Protection Level of Web-keys. The main difference between web-keys and session IDs is that the latter grant complete authority, while web-keys are usually limited in their scope. This reduces the possible impact of leaking web-keys, depending on the situation. As Eselsohr does not use JavaScript, it cannot send tokens with the *HTTP Authorization request header*, which would be suitable to transmit secret data. In a hybrid model, entry points would use web-keys, as they can be used across sites for sharing and can be stored in password or bookmark managers, while in-app navigation would switch to the authorization header.

4.3.3 Sensitive Data Exposure

Instead of trying to attack applications directly, attackers also circumvent security measures by stealing secrets, by performing man-in-the-middle attacks, or getting clear text data directly on the server. Oftentimes sensitive data are encrypted, but this added complexity has its costs: key management is not trivial and choosing weak algorithms or protocols could lead to a false sense of security. Keeping stored data to a minimum is preferable.

Sending secret data in URLs already mandates that some form of transport encryption has to be used. Some URI schemes enforce this, see 6.1 New URI Schemes. Eselsohr can either be deployed behind a reverse proxy, which should handle the encrypted HTTPS communication, or provides an encrypted web server itself. When providing HTTPS directly it uses the recommended *intermediate* TLS configuration by Mozilla [58] as seen in Table 4.1, which does not list any currently known weak cryptographic ciphers, algorithms, or protocols.

TLS Features	Eselsohr
Protocols	TLS 1.2, TLS 1.3
HTTPS Strict Transport Security	max-age=31536000
TLS curves	X25519, P-256, P-384
Ciphers	TLS13_AES128GCM_SHA256 TLS13_AES256GCM_SHA384 TLS13_CHACHA20POLY1305_SHA256 ECDHE_ECDSA_AES128GCM_SHA256 ECDHE_RSA_AES128GCM_SHA256 ECDHE_ECDSA_AES256GCM_SHA384 ECDHE_RSA_AES256GCM_SHA384 ECDHE_ECDSA_CHACHA20POLY1305_SHA256 ECDHE_RSA_CHACHA20POLY1305_SHA256 DHE_RSA_AES128GCM_SHA256 DHE_RSA_AES256GCM_SHA384

Table 4.1: Eselsohr TLS configuration

One primary design goal of Eselsohr was ease of deployment. Therefore, any cryptographic key management was avoided, except for the certificate and key for the TLS connection. This means that persisted data are neither encrypted nor authenticated. The data stored by the prototype—titles of websites—were not determined to be sensitive so no additional protection was implemented. The recommended mode of operation is to self-host Eselsohr; malicious administrators are not part of the threat model. Protecting against those would require the implementation of *End-to-End Encryption* (E2EE) which would increase the complexity of the application significantly.

4.3.4 Broken Access Control

Access control in a web application can be split into two categories: Function level and object level access control. The former allows group x to use endpoint a, while group y does not have this permission, but both can call endpoint b. The latter makes sure that user Alice is allowed to modify objects that were created by her, while objects by Bob are protected from her control.

Function level access control is often implemented by some kind of web framework. Routes have annotations which level or groups of accounts are allowed to access them. For each request the identity of the user is determined—for example, by looking up the session with the provided session ID—and their group association is matched with the allowed groups for that route. If they pass this authorization check, the request is valid and gets performed.

Object level access control on the other hand is implemented by the developers for each application on an individual basis. It is dependent on the specific data structures, increasing the change of making errors. If no proper object ownership is implemented, users can access and modify all data that are available to their group level, a common mistake [59], as long as they know the object identifiers, which could be as simple as increasing integers.

Object capabilities naturally provide access control on the object level. Access is not determined by identity but by possession of capabilities. Users can only access and modify data, which they created themselves, or where capabilities were explicitly shared to allow access. The prototype provides functionality to create shareable access tokens for single entities or for whole collections, while allowing to set permissions on a fine-grained level.

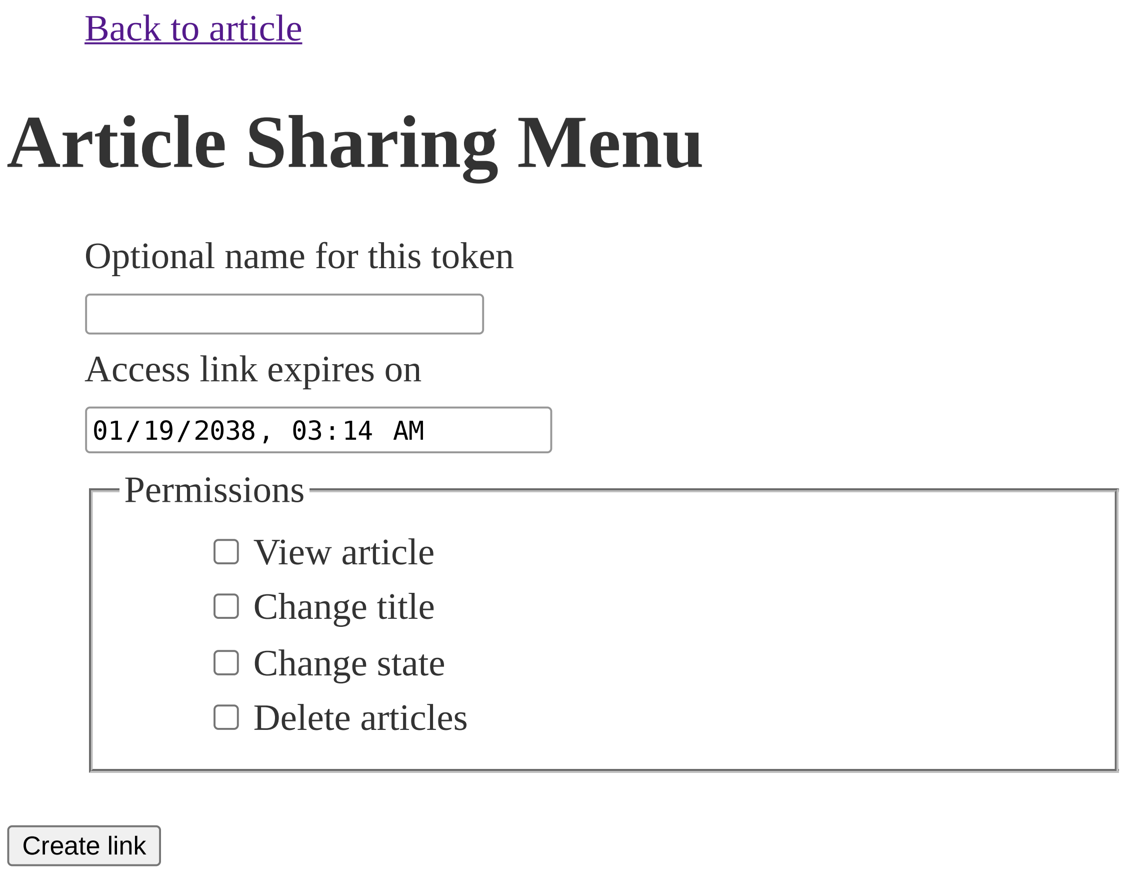

Figure 4.1: Menu for sharing access tokens for article lists in Eselsohr

Figure 4.1 shows how access to the articles list can be shared in Eselsohr. Tokens have an optional name for easier identification for users, can expire, and be revoked at any time. It is also possible to create tokens with restricted permissions, making them suitable to be shared with the public or a limited group of people, depending on the use case. An example for this can be seen in 5.3.4 Embeddability. This is also a demonstration of the deep linking functionality of web-keys as was described in 2.5.1, and how different web applications can cooperate safely with each other.

Type-driven application design provides static guarantees that permission checks are performed for each request. (see 3.1.1 Types as Capabilities) No web controller logic was implemented that allows accessing entities by their ID, thus, access through access tokens is enforced.

4.3.5 Cross-Site Scripting

According to the OWASP, cross-site scripting attacks are found in two thirds of all web applications [3]. User provided input gets interpreted as JavaScript code, which gets executed in the browsers of other users. There are three different types of cross-site scripting attacks: *Reflected*, *Stored*, and *DOM XSS*. The solution to this problem is to correctly encode user input for the target environment. In this case, escaping characters in a way, such that they get not interpreted as HTML but as plain data.

Eselsohr is not susceptible to XSS attacks:

- It is safe against reflected cross-site scripting. Eselsohr only accepts serialized access tokens in an URL. This is checked by explicit type conversions at the boundaries of the application.
- It is safe against stored cross-site scripting. Similar to the enforced authorization checks, static types provide guarantees that user input has to be converted explicitly to an `Html` type, where correct output encoding is applied during the conversion whenever a web page is requested.
- It is safe against DOM cross-site scripting. Eselsohr does not use JavaScript and attacker-controllable data are not dynamically included.

In addition, a strict *Content Security Policy* [60] (CSP) is used as defense-in-depth mitigation. This policy tells browsers to not allow any JavaScript code to be run on the site.

4.3.6 Insecure Deserialization

Deserialization vulnerabilities are often hard to exploit, as they need to be tweaked for the specific target. But the possible impact can be a remote code execution, so these attacks should not be neglected. This class of attack can be split up into two groups: using available classes during the deserialization to perform malicious side-effects, or changing data to gain, for example, administrator privileges.

The prototype utilizes serialization for two use-cases: Access tokens in URLs and for persistence.

Access tokens are serialized Haskell records containing the ID of the resource and ID of the referenced capability.

Listing 4.1: Snippet of Haskell code that is used for serializing access tokens

```haskell
data Reference = Reference
    { resourceId   :: !(Id Resource)
    , capabilityId :: !(Id Capability)
    }
  deriving stock (Generic, Show, Eq)
  deriving anyclass Serialise

newtype Accesstoken =
  Accesstoken {unAccesstoken :: LByteString}
deriving (Eq) via LByteString

instance ToHttpApiData Accesstoken where
  toUrlPiece =
    decodeUtf8 . encodeBase32Unpadded' . unAccesstoken

instance FromHttpApiData Accesstoken where
  parseUrlPiece =
    either
      (const $ Left "invalid UrlToken")
      (Right . Accesstoken)
    . decodeBase32
    . encodeUtf8

mkAccesstoken :: Reference -> Accesstoken
mkAccesstoken = Accesstoken . Ser.serialise

toReference :: Accesstoken -> Reference
toReference = Ser.deserialise . unAccesstoken
```

Listing 4.1 shows the definition of a Haskell record. When used as part of an URL, it gets serialized into a binary string and Base32-encoded. This process is done in reverse when accepting access tokens in requests.

The code shown here is safe against both types of deserialization attacks. As the serialization code is pure, it cannot perform any side-effects. This is enforced by the type system, which makes deserialization safe [50]. The data structure also only contains identifiers, which are then used to lookup the referenced data; so there are no properties that can be manipulated directly.

For persistence, Eselsohr is serializing its data structures so that they can be stored in files. Listing 4.2 shows some relevant Haskell code for this process.

Listing 4.2: Snippet of Haskell code that is used for persisting serialized data

```haskell
encodeFile :: (Serialise a, WithFile env m)
     => FilePath -> a -> m ()
encodeFile fp =
  writeBinaryFileDurableAtomic fp . toStrict . Ser.serialise

decodeFile :: (Serialise a, WithFile env m)
     => FilePath -> m a
decodeFile fp =
  Ser.deserialise . fromStrict <$> liftIO (readFileBS fp)
```

This snippet is also safe against remote code executions, or other side-effect inducing attacks. But it works directly with the data types, which can be manipulated. To reduce complexity and simplify the deployment, Eselsohr does not use cryptographic signatures to authenticate the validity of persisted data. Administrators are advised to keep good server hygiene by using full-disk encryption for data at rest, and to use the provided systemd unit file or Dockerfile to provide sufficient process and file system isolation.

The serialization library in use also states that it is safe to use with untrusted user input, as it is not susceptible to asymmetric resource consumption attacks[3].

4.3.7 Using Components with Known Vulnerabilities

Outdated third-party components are a common attack vector, not only for web applications. Counter measures include keeping dependencies to a minimum, having regular update cycles, or sandboxing components. But modern software projects include thousands of dependencies and sub-dependencies, thus, making it hard to keep track of them all.

Capability-based security projects approach this problem from another direction. Every single package is potentially dangerous, as they run with the same privileges as the program that calls it. Removing these default rights and letting components start with an absolute minimum changes the risk landscape [61].

At first it looks like Haskell's separation of pure and impure code could help us here. Calling code from third-party dependencies that do not run in `IO` cannot have any side-effects, and are therefore safe to call. But once side-effects are allowed, they cannot be limited. There is no way to declare that file system access is forbidden but calling a specific HTTP endpoint is allowed. Also, Eselsohr's mechanism to restrict possible side-effects (see 3.1.2 Types for Explicit Side-effects) cannot be applied to external dependencies. In addition, `unsafePerformIO` can be used to call functions

[3]`https://github.com/well-typed/cborg/blob/9be3fd5437f9d2ec1df784d5d939efb9a85fd1fb/README.md`, Accessed: 2021-08-29

with side-effects in a pure context. There is a subset of Haskell, called *Safe Haskell* [62], that closes these loopholes in the type system, but it is not widely implemented and not in use in Eselsohr.

This is one example where capability-based security would help reducing the danger of a vulnerability class, but its effectiveness is highly dependent on the runtime it is being executed on.

4.3.8 Summary

Table 4.2 shows a summary of our analysis results. We demonstrated that using capability-based techniques mitigates and sometimes even prevent certain vulnerability classes. A comparison was added to the table where we show the protection effects of a pure OCAP system. Capabilities would be protected by a trusted environment, like the operating system or the language runtime, so "Broken Authentication" and "Sensitive Data Exposure" would be no problem anymore. In an OCAP system, the security of the XML parser would depend on the amount of authority passed to the parser, so we scored this class as "mitigated". Certain design choices exist for OCAP systems to prevent issues with deserialization [63]. As mentioned in the corresponding subsection, third-party dependencies in pure OCAP programming languages would start with no authority, so outdated or vulnerable code would cause even less harm.

Vulnerability class	Protection level	
	Eselsohr	Pure OCAP system
A1:2017-Injection	◐	◐
A2:2017-Broken Authentication	◐	●
A3:2017-Sensitive Data Exposure	◐	●
A4:2017-XML External Entities (XXE)	-	◐
A5:2017-Broken Access Control	●	●
A6:2017-Security Misconfiguration	○	○
A7:2017-Cross-Site Scripting (XSS)	●	●
A8:2017-Insecure Deserialization	◐	●
A9:2017-Using Components with Known Vulnerabilities	◐	●
A10:2017-Insufficient Logging & Monitoring	○	○

●=prevention ◐=mitigation; ○=no effect; -=not analyzed;

Table 4.2: Summary of security analysis results

4.4 Protection Level of Web-keys

Unlike object capabilities in a capability-safe programming language, web-keys are not unforgeable. They are also not protected by a runtime system. URL with access tokens are plain strings transmitted over the internet. Anyone who manages to get a hold of a web-key can use it, regardless of whether the token was given intentionally to that person or not. Thus, preventing the leakage of the web-key is of utmost importance.

There are several places where HTTP server and clients can exchange data:

- HTTP header
- HTTP body
- Cookies
- URL
- HTML body

Without the use of JavaScript, a browser can transfer data to the server by:

- fetching a resource over HTTP GET (data can be in the path or in a query string);
- submitting an HTML form over HTTP POST;
- sending a cookie.

JavaScript gives us several more options:

- attaching HTTP header;
- sending data in the HTTP body (commonly used with JSON or XML data types);
- extract data from the URL fragment.

Eselsohr transfers web-keys as query strings inside URLs. As described in 2.5 WATERKEN: WEB-KEY, this has several drawbacks, as browsers do not have a concept of URLs with sensitive data in them. They are stored in server logs and browser caches, can be seen in the browser history, are send by default in the HTTP `referer` header, and can be shoulder surfed.

Table 4.3 shows a comparison of how well protected a web-key is with different transfer methods. The URL path and query string are the most exposed places, while HTTP headers and cookies are the least exposed. But browsers do not support a way that would allow attaching a web-key to a link, where the browser would know that the web-key has to be send as HTTP header or as a cookie. With cookies, explicit control over authority would also be lost.

Transfer method	Extraction method				
	Visible in location bar	Accessible by JavaScript	Send by default in `referer` header	Stored by default in server logs	Stored by default in browser cache
HTTP header	-	◐	-	○	○
HTTP body	-	●	-	○	●
Cookie	-	◐	-	○	●
URL path	●	●	●	●	●
URL query string	●	●	●	●	●
URL fragment	●	●	○	-	●

●=completely exposed; ◐=partly exposed; ○=not exposed; -=not applicable

Table 4.3: Comparison of how different data transfer methods expose web-keys

URL fragments have the benefit of not being send in the referrer header, but require the use of JavaScript. As the behavior of the referrer header can be configured by the HTTP Referrer Policy, the exposure of using the URL path or query string is reduced. The HTML 5 History API can be used via JavaScript to rewrite the content in the location bar, thus, reducing the exposure of the web-key there.

Ideally, browsers would support an URI scheme which addresses the above drawbacks and treat the web-key as secret data. (see 6.1 New URI Schemes)

CHAPTER 5

Model Evaluation

In this chapter, we are going to compare the prototype, built with object capability techniques, against other web applications built on identity-based access-control lists. We are first describing our methodology, present the other contestants, and then show the results of the evaluation.

5.1 Methodology

First, we needed some way to reliably measure indicators that could be used for the upcoming evaluation. *Lines of Code* (LOC) are a questionable metric [64], not only because they have no real meaning, but also because results vary strongly when different programming languages or paradigms are compared. The chosen example projects are not prominent enough to have assigned *Common Vulnerabilities and Exposures* (CVE) so we cannot compare against known security issues that existed in these projects. Therefore, we will compare conceptual differences between ACL-based and OCAP-based applications and use Eselsohr and two other apps for examples.

5.2 Contestants

This section describes two alternatives to Eselsohr, both built on identity-based security paradigms. All these services can be used as a bookmarking server, and serve similar purposes. They are all available under a open-source license and their source code is publicly hosted. This makes it possible to conduct a white-box security analysis.

5.2.1 Wallabag

Wallabag[1] was started in 2013 and is written in PHP[2]. It is built on the Symfony web framework[3]. The Wallabag team offers applications for the web and mobile apps for Android and iOS as well as browser extensions. Wallabag can extract the content of web pages and display it in a more friendly to read format. Several import and export functions are available that help migrating to or from the service. There is also support for RSS feeds, so others can follow the list of stored articles.

5.2.2 Espial

Espial[4] was started in 2019 and is written in Haskell and PureScript[5]. The backend is built on the Yesod web framework[6], while the frontend is written in PureScript[7], a Haskell-like language that transpiles to JavaScript and which is used to build *Single Page Applications* (SPA). Espial users can add, in addition to web pages, also notes with support for Markdown. Articles can not 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. It opens a new browser window with the add form, which has the current page's title and URL already filled in.

5.3 Evaluation Results

For the evaluation, four main aspects were compared. First, we will look at user management, a crucial aspect in identity-based applications. We will then compare how the features of the web applications can be used by analyzing how a privileged endpoint for a single action performs authorization and which other security protections are in place. We will continue by evaluating how resources, which can be accessed by the user, can be shared with others. Finally, we will compare the embeddability of the applications and if parts of them can be integrated within other web applications.

5.3.1 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. This common functionalities are often provided by the web framework in use. Both ACL apps have built their user models on top of code provided by their chosen frameworks.

[1]`https://www.wallabag.it/en`, Accessed: 2021-09-10
[2]Latest commit: `6142adc4dcfef2dda4fe1c244ad8873e5eae71c9`
[3]`https://symfony.com`, Accessed: 2021-09-10
[4]`https://github.com/jonschoning/espial`, Accessed: 2021-09-10
[5]Latest commit: `ef298cfdd0f00c9ebdb0ed56d73b55912a3f03cf`
[6]`https://www.yesodweb.com`, Accessed: 2021-09-10
[7]`https://www.purescript.org`, Accessed: 2021-09-11

The OWASP associates several risks with user management including:

- storing passwords in plain text, in encrypted form, or with a cryptography weak hashing algorithm;
- allowing weak or well-known passwords;
- implementing vulnerable password reset;
- missing brute force protections for the login;
- missing multi-factor authentication.

Because implementing secure user management is a non-trivial task, there is a trend in the industry to externalize it to third-party providers and using *Single Sign-On* (SSO) solutions like *SAML* or *OpenID Connect* for authentication and authorization [65]. 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 temporarily or permanently removed, then these applications can no longer be used or the user has to start over with a new account.

During the analysis of the source code of Wallabag, a security configuration could be found that is not seen as best practice anymore[8]. The project currently uses SHA-512 as password hashing algorithm. This is not ideal as SHA-512 is a comparatively fast operation, a property, which is not desirable for password hashes, as this allows for faster brute forcing attacks. Modern password hash algorithms, like Argon2, provide several mitigations against such attacks and should be used [66]. Espial is using the bcrypt algorithm for password hashes[9], which is suited for this task.

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

5.3.2 Adding a Bookmark

We will now look at how typical ACL-based web applications perform simple data manipulation. In this case, adding a new bookmark is part of a category of operations commonly known as “Create, Read, Update, Delete” (CRUD).

1. The web application accepts a new request from the user.

[8]`https://github.com/wallabag/wallabag/blob/6142adc4dcfef2dda4fe1c244ad8873e5eae71c9/app/config/security.yml#L3`, Accessed: 2021-09-18

[9]`https://github.com/jonschoning/espial/blob/c3a126b9eadb3c3778ab93ed4c4d0e805f669d3c/src/Model.hs#L35`, Accessed: 2021-09-20

2. A routing mechanism maps the URL path from the request to a controller, which handles requests for that particular path.
   - Some frameworks allow to declaratively restrict paths to only certain groups of users.
   - Alternatively, ad-hoc restrictions can be applied in the controller itself.
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, if 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 send 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*[67].

Listing 5.1 shows how adding a new bookmark is handled in Espial[10]. The function **`_handleFormSuccess`** 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.

Listing 5.1: Espial: Controller for adding bookmarks

```
postAddR :: Handler ()
postAddR = do
  bookmarkForm <- requireCheckJsonBody
  _handleFormSuccess bookmarkForm >>= \case
    (Created, bid) -> sendStatusJSON created201 bid
```

[10]https://github.com/jonschoning/espial/blob/c3a126b9eadb3c3778ab93ed4c4d0e805f669d3c/src/Handler/Add.hs#L59, Accessed: 2021-09-20

```haskell
    (Updated, _) -> sendResponseStatus noContent204 ()

_handleFormSuccess :: BookmarkForm -> Handler (UpsertResult,
 ↪  Key Bookmark)
_handleFormSuccess bookmarkForm = do
  (userId, user) <- requireAuthPair
  bm <- liftIO $ _toBookmark userId bookmarkForm
  (res, kbid) <- runDB (upsertBookmark userId mkbid bm tags)
  whenM (shouldArchiveBookmark user kbid) $
    void $ async (archiveBookmarkUrl kbid (unpack
    ↪  (bookmarkHref bm)))
  pure (res, kbid)
  where
    mkbid = BookmarkKey <$> _bid bookmarkForm
    tags = maybe [] (nub . words . T.replace "," " ") (_tags
    ↪  bookmarkForm)
```

The main problem with this kind of controller logic is that the authorization logic is optional. There is no enforcement of access restriction to that endpoint or on objects themselves. In this case, the function `requireAuthPair` is also used for getting data about the currently logged-in user, so it is unlikely that this function gets forgotten, but there are other controllers where that could be the case. Also, if Espial chooses to add a less privileged user group, which does not have permission to create new bookmarks, new authorization checks would have to be added, without any guidance by the compiler or framework.

As seen in 3.1.1 Types as Capabilities and 3.2 URLs as Capabilities, 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 fulfill the complete mediation principle.

5.3.3 Sharing Functionality

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

This is the case in Espial. The bookmarks of users are public by default and are available at `https://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 can also be revoked by the user at any time. It is not possible to generate multiple tokens, or put further restrictions on them. The same token is also used for all available feeds, but it is still a very basic capability system, embedded in an otherwise ACL-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. As mentioned in 4.3.2 Broken Authentication, a hybrid model could be deployed, where only the endpoints that should be shareable are available with web-keys. The rest relies on HTTP header authorization, as long as no other secure option for safe URLs in web browsers exists. (see 6.1 New URI Schemes) Eselsohr also allows to apply more restrictions on web-keys, such as a limited validity period and restricted permission sets.

5.3.4 Embeddability

The inability to embed ACL-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 reduced 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. This works, again, 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 well-meant or malicious intent. This hinders the ability to built 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.

Figure 5.1 shows the Eselsohr page to create a new article, embedded as a custom widget in the instant messenger Element[11]. This messenger allows embedding 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. But web-keys do not have this limitation and, therefore, allow

[11] `https://element.io`, Accessed: 2021-09-05

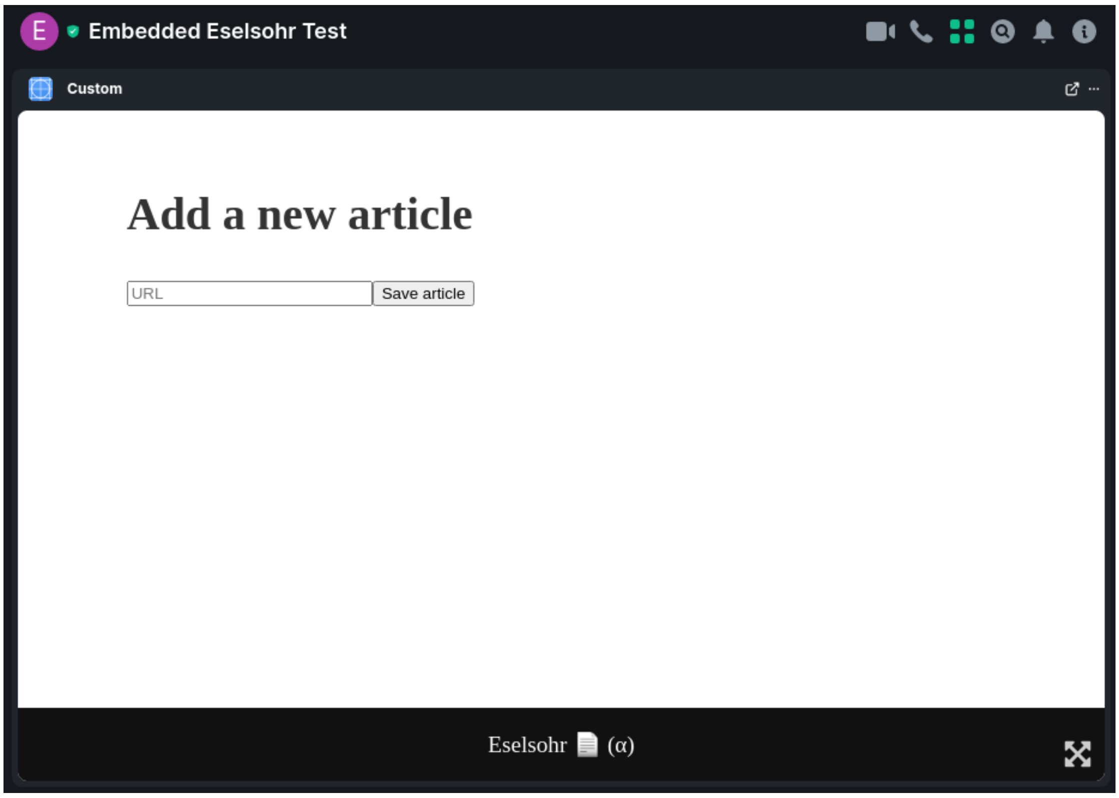

Figure 5.1: Eselsohr page with restricted permissions, embedded in an iframe inside an instant messenger channel

for these scenarios. The provided web-key only has the permission to create new articles for that specific resource and nothing else. If the link would leak, no other actions could be performed with it.

5.3.5 Summary

Table 5.1 answers the question whether security vulnerabilities in authorization systems can be prevented by design. By using types as authorization tokens, we get strong guarantees that authorization checks are not forgotten 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 against vulnerabilities like cross-site request forgery or clickjacking. Using the type system for explicit side-effects improves the reasoning about the code base, as one has 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, like those that handle untrusted user input with deserialization, become secure.

OCAP Technique	Prevented Vulnerability
Types as capabilities	Forgotten authorization checks
Combining designation with authority	Confused Deputy
Explicit side-effect handling	Too much ambient authority

Table 5.1: A list of object capability techniques and what vulnerability they prevent

Table 5.2 answers the question whether OCAP-based applications are at least as secure as ACL-based ones. The chosen properties are the overlying concepts of the techniques that were listed in Table 5.1. In ACL-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 has equal authority. By requiring access tokens within the code, and, thus, making authority explicit, the authorization flow in the program becomes equivalent with the creation and passing of access tokens as arguments. Functions have to explicitly request authority before it can be used.

As subjects in ACL 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, as 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 ACL-based system by creating small identities with minimum rights, but it is hardly practical.

Imagine an administrator with access to several subsystems: the administrator would now have to create several user accounts and groups for themselves, so that they can minimize the authority per user, while always having to check that they do not perform an action with the wrong user. In addition, someone with no administrative privileges might not even be capable of creating new lower privileged accounts. This is not a problem in an object capability system. The fact that someone is an administrator does not prevent them from using capabilities with less privileges for their actions. Also, administrators can share single, high-privileged capabilities with trusted users, without having to put them into a high-privileged, more general admin group.

Property	ACL	OCAP
No Ambient Authority	✗	✓
No Designation without Authority	✗	✓
Principle of Least Authority	∼	✓

Table 5.2: Comparison of security properties of ACL and OCAP

CHAPTER 6

Related Work

Developments in improving authorization schemes and security models are not halting. This chapter presents some noteworthy projects that are occurring in the space of capability-based security.

6.1 New URI Schemes

There are several proposals to introduce new URI schemes which are aware that web-keys are secret data.

The Bearer Capability URI Scheme (Bearcap): Bearcaps [68][69] combine URLs with an access tokens. Listing 6.1 shows an example Bearcap URI. They contain two query parameters: `u` containing the URL associated with the request and `t` containing the access token. When a compliant browser resolves this URI, the token will be send as HTTP Authorization header. Only the URL part will be displayed in the location bar and in other UI elements which would display the URI. The token will also not be send as part of the HTTP Referrer header. Other compliant applications like web servers or proxies would not log the token part by default. This addresses the issues described in Section 4.4.

Listing 6.1: Example Bearcap URI

```
bearcap:?u=https://eselsohr.local/articles/example-title&
t=QMANQJKQCLCDXJT5DITHDRPVMFQF4MN3MCKUBFZVIPHU52S7PX2CI2GU
```

The Bearer URL Scheme: Bearer [70] URLs work, similar to Bearcaps, by combining access token with URLs. Listing 6.2 shows an example Bearer URI. They resemble the formerly supported syntax for HTTP Basic Authentication. In addition to the behavior described for Bearcaps, Bearer compatible browsers can extract the token out of the URIs while building the DOM, making the token inaccessible to JavaScript. Furthermore, the DOM spec could be enhanced, such that passing Bearer URIs to, for example, the fetch API is possible within JavaScript, but accessing the contained token is still forbidden. `UserOnly` is an example for an attribute, similar to cookie attributes, that restricts the use of the token, such that it is only send when triggered by an user gesture, therefore further hardening against XSS attacks.

Listing 6.2: Example Bearer URI

```
bearer://QMANQJKQCLCDXJT5DITHDRPV3MCKUBFZVIPHU52S7PX2CI2GU;
UserOnly@https://eselsohr.local/articles/example-title
```

6.2 GNAP and ZCAP-LD

The *Grant Negotiation and Authorization Protocol* [71] (GNAP), formerly known as OAuth 3.0, is an in-progress next generation protocol. It is based on the experience of practical implementations of OAuth 2 but not compatible with previous OAuth standards. GNAP supports features like:

- requesting multiple access tokens with one grant;
- continuing a grant to, for example, drop privileges when not requiring them anymore;
- a built-in concept for identities, avoiding the need for extensions like OpenID Connect;
- the ability to differentiate between running instances of the same app;
- first-class support for interactions to declare what the authorization server should respond with: a redirect, an app launch, a callback, or no interaction at all if an established side-channel exists, such as carrier pigeons [72].

GNAP is a protocol that can work with the *Authorization Capabilities for Linked Data* [73] (ZCAP-LD) data model. ZCAP-LD combines object capabilities and *Linked Data Proofs* [74] 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.

6.3 Macaroons

Macaroons build up on cookies, hence the name, but extend them with caveats that attenuate or confine them, so they are more suitable for authority delegation purposes[75].

They use nested, chained MACs—HMACs, to be more precise—to append restrictions to the cookies, like for what they can be used for, when usage is allowed, if additional authentication proofs are required, such as third-party signatures, and more. Each appended caveat is an element of a list of predicates that all have to be valid, when they are evaluated for each particular request.

6.4 Modern OCAP Developments

The OCAP community continues the steadily improvement of this security model and works on new projects and collaborates with committees to standardize these techniques. Some of these newer developments are listed here.

6.4.1 Endo

Endo is a sandboxed and OCAP-safe subset of JavaScript developed by Agoric [76]. It offers protection against malicious third-party dependencies by explicitly passing powerful capabilities, like the ability to interact with the network. Reviewable policies allow to reduce the amount of authority to a minimum.

As part of the overall project, a new ECMAScript standard called *ShadowRealm* is currently proposed that would eliminate the risks of cross-site scripting attacks [77]. This is done by running user-provided input in an environment void of capabilities. The attack surface is reduced without access to the network, the file system, or the console.

Agoric also works on implementing a JavaScript version of a capability transport protocol as part of their platform SDK[1].

6.4.2 The Spritely Project

Spritely uses object capabilities to build a platform for federated social networks [78]. It allows secure distributed computing and is implemented in Racket, a LISP-like programming language. The project was heavily inspired by the works of Miller et. al. and closely works with Agoric to be compatible with their new CapTP implementation. The project's components each handle a specific area of application development, like storage and file distribution, identity management, serialization, and debugging, while implementing state of the art OCAP techniques.

[1]`https://github.com/Agoric/agoric-sdk/blob/5521c6475242d8835acf71dda4311e5a55b4c23e/packages/captp/README.md`, Accessed: 2021-08-31

The CapTP implementation in Spritely, the foundational layer for Spritely, supports different transport layer protocols and can be used, for example, over the Tor network to allow for peer-to-peer applications [79].

6.4.3 Cap'n Proto

Cap'n Proto [80] is a serialization format and RPC framework, suitable for the secure exchange of data. It utilizes capability-based security and can be seen as capability transfer protocol.

The design of Cap'n Proto was based on the experiences the maintainer made when developing another serialization protocol called *Protocol Buffers* (ProtoBuf). Unlike in ProtoBuf, the encoding of Cap'n Proto messages is suitable as a data interchange format as well as an in-memory representation, so no separate encoding and decoding step is required. The encoding is also platform independent and was optimized for modern CPUs. Promise pipelining is supported as well and future integration of the Noise Protocol Framework is planned [81], a cryptographic framework already used by other projects like the WireGuard VPN [82].

Originally created for sandstorm.io [83], it is now developed by a team at Cloudflare, primarily in use in their serverless computing platform [84].

6.4.4 Extending the Prototype

The current system has still potential for improvements, as it is missing several features found in advanced web applications, such as providing an API for external clients, integration with external services, or applying encryption for confidentiality and integrity purposes.

The backend part of Eselsohr could be extended to provide access to external clients via an API. When using the API over HTTP, *Hypermedia as the Engine of Application State* [85] (HATEOAS) could then be used to communicate available capabilities to clients. These external clients could also use more advanced techniques to securely exchange authorization tokens, like Cap'n Proto. This protocol could also be used for inter-process communication, in case the prototype migrates to a multi-process or decentralized architecture. Alternatively, a custom implementation of Bearcaps or Bearer URIs could be used, when direct support by browsers is not required. This was, for example, already implemented in the federated social network Mastodon [86].

Support for ATOM [87] as a syndication format and protocol could be used to further demonstrate, how authorized URLs in form of web-keys can integrate with existing software without them having to implement custom authorization code.

The prototype omitted the implementation of authentication on purpose to avoid the possible confusion that authorization is tied to authentication. But this does not mean that authentication, or a concept of identity, is useless. A future version of the prototype could add support for users and authentication to implement functionality like "Audit log: User x created token y", while still handling authorization with access tokens. This could also be used to distribute capabilities between users in the system, without having them to rely on secure side-channels to exchange tokens with each other.

CHAPTER 7

Conclusion

This thesis proposes an alternative authorization model for web applications. Object capabilities combine references to objects with the associated set of permissions. We showed what 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 could then demonstrate that programming in an object capability style helps eliminating certain security vulnerability classes on an architectural level, and provided some techniques and patterns based on this style, like web-keys.

A functional prototype was implemented to demonstrate that these techniques can be used in practice. The security analysis and model evaluation showed that OCAP-based applications have no significant drawbacks compared to ACL-based applications, while providing improvements in areas like 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 to be the transfer of capabilities over URLs. Hyperlinks are the number one method to navigate between web applications, but web browsers currently assume that URLs only contain non-sensitive information, making it hard to embed secrets like capabilities. In addition, a capability in an URL is a plain string with no further protection, so anyone could come up with a possibly valid capability, although it was not passed to them explicitly.

The prototype was developed in a programming language, which was not explicitly designed for this style of programming. It is capable of running 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 over current OCAP related projects was given and how the prototype could be further improved in the future.

7.1 Future Work

For future research, it should be evaluated how existing features in web browsers could be used to circumvent the current limitations in regard to transferring capabilities. This thesis focused on 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, 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.

Listings

List of Figures

List of Tables

Bibliography

[1] K. Smith, A. Jones, L. Johnson, and L. Smith, "Examination of cybercrime and its effects on corporate stock value," *Journal of Information, Communication and Ethics in Society*, 03 2019.

[2] K. Mlitz, "Size of the cybersecurity market worldwide, from 2021 to 2026." `https://www.statista.com/statistics/595182/worldwide-security-as-a-service-market-size/`, 08 2021. Online; Accessed: 2021-09-25.

[3] I. OWASP Foundation, "Owasp top ten 2017." `https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/`, 2017. Online; Accessed: 2021-09-25.

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

[5] H. M. Levy, *Capability-Based Computer Systems.* USA: Butterworth-Heinemann, 1984.

[6] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, p. 143–155, Mar. 1966.

[7] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: a fast capability system," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, p. 170–185, 12 1999.

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

[9] K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, p. 133–150, 2013.

[10] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *Proceedings of the 4th ACM international workshop on Storage security and survivability*, p. 21–26, 2008.

[11] G. Steed and S. Drossopoulou, “A principled design of capabilities in pony,” *Master’s thesis, Imperial College*, 2016.

[12] C. Morningstar, “What are capabilities?.” `http://habitatchronicles.com/2017/05/what-are-capabilities/`, 05 2017. Online; Accessed: 2021-09-09.

[13] J. A. Rees, “A security kernel based on the lambda-calculus,” *A. I. MEMO 1564, MIT*, vol. 1564, 1996.

[14] R. C. Martin, “Getting a solid start.” `https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start`, 12 2009. Online; Accessed: 2021-09-09.

[15] M. S. Miller, K.-P. Yee, J. Shapiro, *et al.*, “Capability myths demolished,” tech. rep., 12 2003.

[16] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Reading, MA: Addison-Wesley, 1995.

[17] B. W. Lampson, “Protection,” *SIGOPS Oper. Syst. Rev.*, vol. 8, p. 18–24, Jan. 1974.

[18] D. Ferraiolo and R. Kuhn, “Role-based access control,” in *In 15th NIST-NCSC National Computer Security Conference*, p. 554–563, 10 1992.

[19] V. Hu, D. Kuhn, and D. Ferraiolo, “Attribute-based access control,” *Computer*, vol. 48, pp. 85–88, 02 2015.

[20] T. Close, “Acls don’t.” `https://papers.agoric.com/assets/pdf/papers/acls-dont.pdf`, 2009.

[21] N. Hardy, “The confused deputy: (or why capabilities might have been invented),” *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, p. 36–38, 1988.

[22] E. Communities, “The e extensions to java.” `http://erights.org/history/original-e/e/e_white_paper.html`, 1996. Online; Accessed: 2021-09-22.

[23] P. Seibel, *Coders at Work: Reflections on the Craft of Programming*. Apress, 12 2009.

[24] C. Morningstar and F. R. Farmer, “The lessons of lucasfilm’s habitat,” in *Cyberspace: First Steps*, MIT Press, 1990. Online; Accessed: 2021-09-22.

[25] A. Mettler, D. Wagner, and T. Close, “Joe-e: A security-oriented subset of java,” in *Network and Distributed Systems Symposium*, Internet Society, 2010.

[26] C. Simpson, “Object capability discipline.” `https://monte.readthedocs.io/en/latest/intro.html#object-capability-discipline`, 12 2018. Online; Accessed: 2021-09-25.

[27] M. S. Miller, *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[28] M. Stiegler and M. S. Miller, "A capability based client: The darpabrowser," *Technical Report Focused Research Topic*, 2002.

[29] M. Stiegler, A. Karp, K.-P. Yee, T. Close, and M. Miller, "Polaris: Virus-safe computing for windows xp," *Commun. ACM*, vol. 49, pp. 83–88, 09 2006.

[30] C. Lemmer-Webber, "What is captp, and what does it enable?." `https://spritelyproject.org/news/what-is-captp.html`, 02 2021. Online; Accessed: 2021-09-07.

[31] C. Lemmer-Webber, "Goblins: a transactional, distributed actor model environment." `https://docs.racket-lang.org/goblins/index.html`, 2021. Online; Accessed: 2021-09-07.

[32] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), p. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[33] M. S. Miller, E. Tribble, and J. Shapiro, "Concurrency among strangers," in *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, vol. 3705 of *Lecture Notes in Computer Science*, pp. 195–229, 04 2005.

[34] M. S. Miller, T. Van Cutsem, and B. Tulloh, "Distributed electronic rights in javascript," in *Proceedings of the 22nd European conference on Programming Languages and Systems*, vol. 7792, pp. 1–20, 03 2013.

[35] J. Armstrong, *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 12 2003.

[36] M. S. Miller, "Promise pipelining." `http://www.erights.org/elib/distrib/pipeline.html`, 10 1998. Online; Accessed: 2021-09-07.

[37] M. S. Miller, "Captp: Distributed acyclic garbage collection." `http://erights.org/elib/distrib/captp/dagc.html`, 10 1998. Online; Accessed: 2021-09-07.

[38] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), p. 75–88, Association for Computing Machinery, 2008.

[39] S. Lekies, M. Heiderich, T. Holz, and M. Johns, "On the fragility and limitations of current browser-provided clickjacking protection schemes," in *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, (Bellevue, WA), USENIX Association, Aug. 2012.

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

[41] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1." RFC 2616, 06 1999. Online; Accessed: 2021-09-25.

[42] J. Eisinger and E. Stark, "Referrer policy." `https://www.w3.org/TR/referrer-policy/`, 01 2017. Online; Accessed: 2021-09-05.

[43] D. Hardt, "The oauth 2.0 authorization framework." RFC 6749, 10 2012. Online; Accessed: 2021-09-25.

[44] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," *CoRR*, vol. abs/1601.01229, 2016.

[45] M. Jones and D. Hardt, "The oauth 2.0 authorization framework: Bearer token usage." RFC 6750, 10 2012. Online; Accessed: 2021-09-25.

[46] D. Hardt, A. Parecki, and T. Lodderstedt, "The oauth 2.1 authorization framework," Internet-Draft draft-ietf-oauth-v2-1-02, Internet Engineering Task Force, 03 2021. Online; Accessed: 2021-09-25.

[47] T. Lodderstedt, J. Richer, and B. Campbell, "Oauth 2.0 rich authorization requests," Internet-Draft draft-ietf-oauth-rar-05, Internet Engineering Task Force, 05 2021. Online; Accessed: 2021-09-25.

[48] S. Peyton Jones, "A history of haskell: being lazy with class," in *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, 06 2007.

[49] G. Bernhardt, "Functional core, imperative shell." `https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell`, 07 2012. Online; Accessed: 2021-09-25.

[50] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in java," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), p. 161–174, Association for Computing Machinery, 2008.

[51] S. Wlaschin, "Using types as access tokens." `https://fsharpforfunandprofit.com/posts/capability-based-security-3/`, 01 2015. Online; Accessed: 2021-09-25.

[52] T. Hoare, "Null references: The billion dollar mistake." https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, 08 2009. Online; Accessed: 2021-09-25.

[53] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[54] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," 08 1997.

[55] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[56] P. Wadler, "Monads for functional programming," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, (Berlin, Heidelberg), p. 24–52, Springer-Verlag, 1995.

[57] K. Townsend, "Credential stuffing: a successful and growing attack methodology." https://www.securityweek.com/credential-stuffing-successful-and-growing-attack-methodology, 2017. Online; Accessed: 2021-08-28.

[58] A. King and J. Vehent, "Security/server side tls." https://wiki.mozilla.org/Security/Server_Side_TLS#Intermediate_compatibility_.28recommended.29, 2021. Online; Accessed: 2021-08-28.

[59] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* USA: McGraw-Hill, Inc., 1 ed., 2009.

[60] M. West, A. Barth, and D. Veditz, "Content security policy level 2." https://www.w3.org/TR/CSP2/, 12 2016. Online; Accessed: 2021-08-29.

[61] K. Sills, "Pola would have prevented the event-stream incident." https://medium.com/agoric/pola-would-have-prevented-the-event-stream-incident-45653ecbda99, 12 2018. Online; Accessed: 2021-08-29.

[62] S. Peyton Jones, "Safe haskell," in *Haskell '12: Proceedings of the Fifth ACM SIGPLAN Symposium on Haskell*, ACM, September 2012.

[63] M. S. Miller, "Safe serialization under mutual suspicion." http://erights.org/data/serial/jhu-paper/index.html, 10 1998. Online; Accessed: 2021-09-25.

[64] E. W. Dijkstra, "On the cruelty of really teaching computing science." https://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html, 12 1988. Online; Accessed: 2021-09-10.

[65] T. Bazaz and A. Khalique, “A review on single sign on enabling technologies and protocols,” *International Journal of Computer Applications*, vol. 151, p. 18–25, 10 2016.

[66] J. Wetzels, “Open sesame: The password hashing competition and argon2,” *CoRR*, vol. abs/1602.03097, 2016.

[67] C. Michael, M. Gegick, and S. Barnum, “Complete mediation.” `https://us-cert.cisa.gov/bsi/articles/knowledge/principles/complete-mediation`, 09 2005. Online; Accessed: 2021-09-20.

[68] A. Conill, “The bearer capability uri scheme.” `https://git.sr.ht/~kaniini/draft-conill-bearcaps-uri-scheme/tree/22c458a95992e56ac41f1fff745855b14a811046/item/draft-conill-bearcaps-uri-scheme.txt`, 2019. Online; Accessed: 2021-08-05.

[69] C. Lemmer-Webber, “Bearcap uris.” `https://github.com/cwebber/rwot9-prague/blob/908c5522720f0e3debad2c1578c28a984660ba05/topics-and-advance-readings/bearcaps.md`, 2019. Online; Accessed: 2021-08-05.

[70] N. Madden, “Towards a standard for bearer token urls.” `https://neilmadden.blog/2021/03/20/towards-a-standard-for-bearer-token-urls/`, 2021. Online; Accessed: 2021-08-05.

[71] G. working group, “Grant negotiation and authorization protocol.” `https://datatracker.ietf.org/doc/charter-ietf-gnap/01/`, 07 2020. Online; Accessed: 2021-08-30.

[72] D. Waitzman, “Ip over avian carriers with quality of service.” RFC 2549, 04 1999. Online; Accessed: 2021-09-25.

[73] C. Lemmer-Webber, M. Sporny, and M. S. Miller, “Authorization capabilities for linked data v0.3.” `https://w3c-ccg.github.io/zcap-ld/`, 12 2020. Online; Accessed: 2021-08-31.

[74] D. Longley and M. Sporny, “Linked data proofs 1.0.” `https://w3c-ccg.github.io/ld-proofs/`, 06 2021. Online; Accessed: 2021-08-31.

[75] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, “Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud,” in *Network and Distributed System Security Symposium*, 02 2014.

[76] Agoric, “Endo.” `https://github.com/endojs/endo`, 2021. Online; Accessed: 2021-08-31.

[77] Agoric, "Ecmascript spec proposal for shadowrealm api." `https://github.com/tc39/proposal-shadowrealm`, 2021. Online; Accessed: 2021-09-25.

[78] C. Lemmer-Webber, "Spritely: Social worlds await." `https://spritelyproject.org`, 2021. Online; Accessed: 2021-08-31.

[79] C. Lemmer-Webber, "Spritely goblins v0.8 released!." `https://spritelyproject.org/news/goblins-0.8.html`, 07 2021. Online; Accessed: 2021-09-24.

[80] K. Varda, "Cap'n proto: Introduction." `https://capnproto.org/`, 2021. Online; Accessed: 2021-09-25.

[81] K. Varda, "Cap'n proto: Road map." `https://capnproto.org/roadmap.html`, 2017. Online; Accessed: 2021-09-24.

[82] J. Donenfeld, "Wireguard: Next generation kernel network tunnel," in *NDSS Symposium 2017*, 01 2017.

[83] K. Varda, "Sandstorm security non-events." `https://docs.sandstorm.io/en/latest/using/security-non-events/`, 10 2016. Online; Accessed: 2021-09-25.

[84] K. Varda, "The sandstorm team is joining cloudflare." `https://sandstorm.io/news/2017-03-13-joining-cloudflare`, 03 2017. Online; Accessed: 2021-08-30.

[85] R. T. Fielding, *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[86] A. Tarawneh, "Bearcaps." `https://docs.joinmastodon.org/spec/bearcaps/`, 12 2020. Online; Accessed: 2021-09-24.

[87] B. de hÓra and J. Gregorio, "The atom publishing protocol." RFC 5023, 10 2007. Online; Accessed: 2021-09-25.

Appendix

Eselsohr will be published under the *European Union Public License*[1] (EUPL) at the following public repository: `https://github.com/mkoppmann/eselsohr`.

The hash of the latest Git commit at the time of submitting this thesis is: `92ffe2005d8acf097e42ec482369e514b471e78a`.

[1] `https://joinup.ec.europa.eu/collection/eupl/eupl-guidelines-faq-infographics`, Accessed: 2021-09-20