Trust Chain
Second Call for Proposals

The WAC Machine

Decentralised, provably secure and consistent Web Access Control

Contents

1. Project Summary 1
2. Applicant Background 2
3. Detailed Proposal Description 3
3.1. Concept and Objectives 3
3.2. Proposed Solution 5
3.3. Expected Impact 8
3.4. Business Model and Sustainability 8
3.5. Implementation 8
References 9

1. Project Summary

wada wada

2. Applicant Background

wada wada

3. Detailed Proposal Description

3.1. **Concept and Objectives.** The past two decades have witnessed the rise of online, user-centred services, most notably social media, which has resulted in a staggering amount of user-generated content being produced and stored online. However, individuals have seldom retained true ownership and control of their own online data. Instead, few large corporations have been amassing disproportionate amounts of user data, leaving individuals with little to no control over how their data are shared, augmented and processed to generate revenue. This state of affairs has raised serious concerns about online data, ranging from the manifest, such as security and privacy, to the subtle and hotly debated ethical and societal implications.[12]

Can individuals regain control of their own online data? Although it appears to be a problem of gargantuan proportions, online data privacy and security can and must be improved. Indeed, the inventor of the Web himself, Tim Berners-Lee, has embarked on a journey to address this very problem with the Solid project[11](https://solidproject.org/) under the auspices of the W3C. Solid aims to repair the Web by enabling true data ownership and strengthened privacy through distributed, decentralised applications controlled by individuals rather than large corporations.

We intend to contribute to the Solid journey by improving and extending Web Access Control (WAC), a core component of Solid's privacy and security architecture. For individuals to truly regain control of their online data in a decentralised architecture, strong correctness guarantees are needed about the software implementing the WAC specification. (Given that software is plagued with defects, why should a user trust an implementation to be secure?) However, to ascertain whether an implementation is correct, the specification must be unambiguous, i.e., exactly one interpretation exists, and consistent, i.e., free of contradictions.

Undoubtedly a remarkable achievement, the WAC specification nonetheless, by dint of being expressed in plain English, suffers from ambiguity and consistency issues. Such issues may jeopardise privacy and security in a decentralised scenario where end-users share and/or migrate data across servers and/or providers. To see why that may be the case, consider WAC's resource hierarchies and access control policies (ACL). The specification is not clear about exactly what constitutes a hierarchy and states that a resource inherits its ACL policy from its container. Now a conceivable resource hierarchy could be given by a resource r owned by Alice and shared among two different containers c_1 and c_2. For the sake of argument, assume policy p_1 is attached to c_1, stating that user u can read. Similarly a policy p_2 is attached to c_2 and allows u to delete. Futhermore, suppose two server implementations exist s_1 and s_2 each with a different interpretation of how r should inherit its policy from the parent container: s_1 selects c_1 as a parent whereas s_2 chooses c_2. Alice keeps r on s_1 and knows from experience with how s_1 enforces security that u is not able to delete r. However after migrating r to s_2, to Alice's dismay, u deletes r.

This proposal aims to achieve the following objectives.

Formal specification: Make the WAC specification unambiguous and consistent so to be able to produce implementations that can be trusted to be secure and correct; Extend the specification to cater to data sharing through advanced cryptography

which allows to selectively disclose only some parts of the data or have algorithms process encrypted data without revealing any actual content.

Access control policies: A domain-specific language to express access control rules in a language close to plain English.

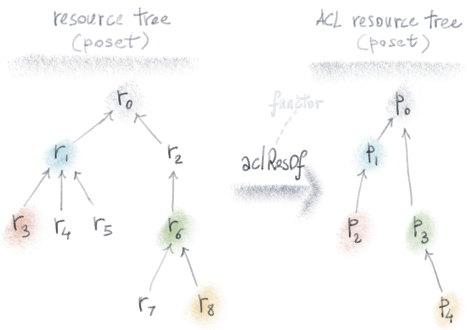
Decentralised, trusted access control: Decentralised network to empower end-users to share and migrate their data and policies among servers without the risk of data breaches due to different WAC implementations interpreting the specification differently.

3.2. **Proposed Solution.** We now turn our attention to the technical details of the proposed solution. We address how to improve and extend the WAC specification to produce software that can be trusted to be secure and correct. We then explain how a decentralised network of such software empowers end-users to share and migrate their data and ACL policies among servers without the risk of data breaches due to different WAC implementations interpreting the specification differently. The novelty of the approach lies in the adoption of mathematical techniques to ultimately produce software that users can trust to be correct. We have an embryonic implementation of the system presented below (TRL 2/3; http://github.com/c0c0n3/whack-wac) which we would like to develop further to reach TRL 6 or 7.

3.2.1. *Formal specification.* The formalisation and extension of the WAC specification will be attained through subsequent refinement phases. In the tradition of functorial semantics[8, 1], a preliminary algebraic model will provide a concise, consistent and unambiguous interpretation of WAC in terms of basic structures and structure-preserving transformations in the elementary theory of the categories of sets[7, 9]. We will then engage with the WAC authors on GitHub to validate and refine the model. Following that, we will devise a specification extension to make the evaluation model more flexible so as to allow the evaluation of access control policies expressed in terms of predicates on a generic set—i.e., functions from a set X to the Boolean algebra $\{0, 1\}$. At the same time, we will investigate another specification extension to accommodate data sharing through functional[2] and homomorphic[5] encryption techniques. Finally, we will encode the formal specification in an advanced programming language (either Haskell[10] or Idris[3]) which we will then leverage to produce a machine-checked proof of correctness of the resulting computer program. The resulting (correct!) executable specification will be submitted to the Solid project for consideration as a future version of WAC.

To illustrate our approach, consider WAC's authorisation process (§5) and ACL resource discovery (§3.1). One possible interpretation of the specification is that a server arranges information resources in a tree $\mathtt{ResTree}$ and, likewise, maintains an ACL resource tree $\mathtt{ACLResTree}$ containing the policies that protect the resources in $\mathtt{ResTree}$. Neither tree is empty. Now a tree is the same as a poset with a terminal object. Each child-parent edge is a morphism $\mathtt{child} \to \mathtt{parent}$ and the tree's root node is then terminal.

On receiving an HTTP request, the server determines the ACL resource that protects the resource which the request targets. We model this lookup procedure as a functor $\mathtt{aclResOf} : \mathtt{ResTree} \to \mathtt{ACLResTree}$. Thus, each path $t_0 \to t_1 \to \ldots \to t_m$ in $\mathtt{ResTree}$ goes to a path $u_0 \to u_1 \to \ldots \to u_n$ in $\mathtt{ACLResTree}$ with $n \leqslant m$. Then if r is the resource that the incoming HTTP request targets, $\mathtt{aclResOf}(r)$ is its corresponding ACL resource. The figure on the right depicts an example $\mathtt{aclResOf}$ functor. Colours hint how the functor maps resource nodes to ACL resource nodes: r_0, r_1, r_3, r_6 and r_8 are mapped, respectively, to p_0, p_1, p_2, p_3 and p_4. Since paths must go to paths, r_4 and r_5 are forced to map to p_1. Ditto



for r_7 that must go to p_3. Finally, r_2 could either map to p_0 or p_3, but in our example $aclResOf(r_2) = p_0$.

To translate an arbitrary $aclResOf$ functor into e.g. Haskell code, observe that such a functor can be defined in terms of the function $path$ which finds the path from the root of the tree to a node satisfying a given predicate p. In turn, $path$ can easily be defined on a canonical multi-way tree structure with the help of the standard list processing function $concatMap$. It is just as easy to prove $path$ correct by induction. Below is the corresponding Haskell code.

```
data Tree a = Node a [Tree a]

path :: (a -> Bool) -> Tree a -> [a]
path p = collect []
  where
  collect xs (Node a ts) | p a       = a:xs
                         | otherwise = concatMap (collect (a:xs)) ts
```

In conclusion, we have turned an informal, ambiguous specification into a precise model that can be reasoned about mathematically and have encoded it into a program which can be proved to satisfy the specification. Crucially, functoriality captures WAC's idea of ACL resource inheritance. For example, with reference to the figure above, r_7 "inherits" r_6's ACL resource, p_3, because the functor preserves paths, hence, necessarily, $aclResOf(r_7) = p_3$. Thus, the lengthily WAC discussion about effective ACL resources and ACL resource discovery can be distilled into the concise statement that there is a given functor $aclResOf : ResTree \to ACLResTree$.

3.2.2. *Access control policies.* As mentioned earlier, access control policies can be expressed in terms of predicates. In fact, the Boolean algebra operations on $\mathbb{B} = \{0, 1\}$ are readily lifted to any function space $\mathbb{B}^X$ by point-wise definition—e.g., for any $p, q \in \mathbb{B}^X$ define $p \wedge q$ as $(p \wedge q)(x) = p(x) \wedge q(x)$. We will exploit this fact to design a domain-specific language embedded in Idris or Haskell (EDSL[6]) which allows policy authors to express access control rules in a language close to plain English. For example, to state that an administrator can read or write a Web resource whereas anyone named Joe and born after 1995 is allowed to perform any operation on that resource, the policy author would write something similar to the following

```
role admin can read or can write
or anyone who has name (== "joe"), has dob (> 1995) can do anything
```

The policy interpreter parses the above into predicates on a given set X (typically the data contained in a Web request, which, in this case, the policy author expects to have $name$ and dob fields) and operations on $\mathbb{B}^X$, then uses these operations to combine and evaluate the predicates. Note that the policy interpreter is embedded, thus the user benefits from the read-eval-print loop (REPL) of the host language for developing and testing policies interactively.

The policy-as-a-predicate WAC extension mentioned in the previous section will enable seamless integration of our EDSL in the Solid ecosystem. End-users will be able to express access control policies in a more concise, direct and natural way than it is currently possible with ODRL or Access Control Policy—the go-to RDF ontologies for ACL policies in Solid. For comparison purposes, expressing the same example EDSL policy as earlier in ODRL

would require a page of terse Turtle as it can be evinced from the following snippet which only encodes the small fragment $\mathtt{dob}(> 1995)$

```
@base <http://example.com/> .
@prefix ex: <http://example.com/> .
@prefix odrl: <http://www.w3.org/ns/odrl/2/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:dob a odrl:Constraint ;
    odrl:leftOperand odrl:dateTime ;
    odrl:operator odrl:lt ;
    odrl:rightOperand "1995"^^xsd:date .
```

3.2.3. *Decentralised, trusted access control.* Our machine-checked, executable specification will enable certifying that a Solid server (pod) implements the formal WAC specification correctly. In particular, this guarantees that all certified server implementations evaluate ACL policies consistently, in the exact same way. With such a guarantee in place, users are able to share and migrate their data and ACL policies among any number of certified servers without the risk of data breaches due to different implementations interpreting the specification differently.

The certification protocol entails property testing[4] and is grounded in the distributed identity software developed during the first round of open calls—e.g., DidRoom. The property testing tool attempts to find counterexamples of the algebraic laws of the specification which are thought to hold true by randomly choosing a large amount of data points according to a suitable probability distribution. The most basic property is that, given the same inputs, the executable specification's output and that of the server under certification agree. If the tool finds no flaws, entries are added to the distributed ledger to record the hashes of the software involved in the process (executable specification, server under certification, generated data points, etc.) and then a DID is assigned to the server which resolves to a DID document containing a specific property that witnesses the successful certification. An end-user interested in high-assurance, certified Solid servers would then choose one based on whether the server's DID document indicates successful certification.

3.3. **Expected Impact.** wada wada

3.4. **Business Model and Sustainability.** wada wada

3.5. **Implementation.** wada wada

3.5.1. *Deliverables and milestones.* wada wada

References

[1] Filippo Bonchi, Dusko Pavlovic, and Pawel Sobocinski. "Functorial semantics for relational theories". In: *arXiv preprint arXiv:1711.08699* (2017).

[2] Dan Boneh, Amit Sahai, and Brent Waters. "Functional encryption: Definitions and challenges". In: *Theory of Cryptography: 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings 8*. Springer. 2011, pp. 253–273.

[3] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of functional programming* 23.5 (2013), pp. 552–593.

[4] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, pp. 268–279.

[5] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[6] Jeremy Gibbons and Nicolas Wu. "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 2014, pp. 339–347.

[7] F William Lawvere. "An elementary theory of the category of sets". In: *Proceedings of the national academy of sciences* 52.6 (1964), pp. 1506–1511.

[8] F William Lawvere. "Functorial semantics of algebraic theories". In: *Proceedings of the National Academy of Sciences* 50.5 (1963), pp. 869–872.

[9] Tom Leinster. "Rethinking set theory". In: *The American Mathematical Monthly* 121.5 (2014), pp. 403–415.

[10] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003. ISBN: 0521826144. URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0521826144.

[11] Andrei Vlad Sambra et al. "Solid: a platform for decentralized social applications based on linked data". In: *MIT CSAIL & Qatar Computing Research Institute, Tech. Rep.* (2016).

[12] Nicole Stremlau, Iginio Gagliardone, and Monroe Price. "World trends in freedom of expression and media development 2018". In: *World Trends in Freedom of Expression and Media Development* (2018).