Linked Data Capabilities

A White Paper from Rebooting the Web of Trust V

by Christine Lemmer-Webber and Mark S. Miller

OVERVIEW

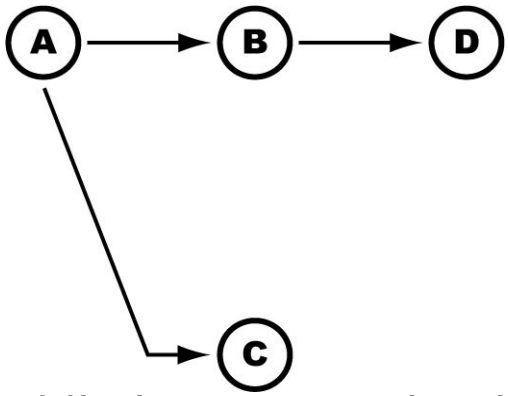
Linked Data Signatures enable a method of asserting the integrity of linked data documents that are passed throughout the web. The object capability model is a powerful system for ensuring the security of computing systems. In this paper, we explore layering an object capability model on top of Linked Data Signatures via chains of signed proclamations[1]. We call this system "Linked Data Capabilities", or "ld-ocap" for short.

The system we propose can work regardless of whether https identifiers or DIDs are being used. Since DIDs work nicely with this system and add an additional layer of decentralization we use them for the URIs of this system.

EXAMPLE SCENARIO

Alice (A) has a direct capability to store files in a "Cloud Storage" system (C). She would like to share this capability with Bob (B), but she is wary of Bob's fondness of storing high-resolution video, so she would like to add a constraint that he may only upload files that are no larger than 50 megabytes at a time. Bob is excited to take advantage of this service because he has recently been playing with Dummy Bot (D), which automatically uploads some photos now and then. But Bob has heard mixed reviews of Dummy Bot and is worried that maybe Dummy Bot will malfunction. He has decided that a 30-day window is a sufficient trial period for permitting Dummy Bot to upload to the storage system, so that he can determine whether to renew at some future date.

The initial condition looks like this:



(A)lice has a capability to the (C)loud storage system through which she can upload

1 What we are calling "proclamations" have also been called "certificates" in previous work such as SPKI and CapCert, we have chosen the name "proclamation chain" to make clear that the structure we are proposing holds none of the centralization traditionally associated with "certificate authorities".

files. (A)lice also has a capability to send a message to (B)ob, and (B)ob has a capability to send a message to (D)ummy Bot.

Each of these characters has an associated linked data document that represents them within the system, making use of JSON-LD and Linked Data Signatures.

Here is Alice:

```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 // This is a DID, but it could as well be an https: uri
 "id": "did:example:83f75926-51ba-4472-84ff-51f5e39ab9ab",
 // This object is a person named Alice
 "type": "Person",
 "name": "Alice",
 // Finally, a signature verification key Alice will be using
 // for her upload capability to the Cloud Storage system
 "publicKey": [{
   // This has its own separate id because it is technically
   // a separate document
   "id": "did:example:83f75926-51ba-4472-84ff-51f5e39ab9ab#key-1",
   "owner": "did:example:83f75926-51ba-4472-84ff-51f5e39ab9ab",
   "publicKeyPem": "-----BEGIN PUBLIC KEY-----\r\n..."}]}
```

Here is Bob:

```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 "id": "did:example:ee568de7-2970-4925-ad09-c685ab367b66",
 "type": "Person",
 "name": "Bob",
 "publicKey": [{
   "id": "did:example:ee568de7-2970-4925-ad09-c685ab367b66#key-1",
   "owner": "did:example:ee568de7-2970-4925-ad09-c685ab367b66",
   "publicKeyPem": "-----BEGIN PUBLIC KEY-----\r\n..."}]}
```

Here is Dummy Bot:

```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 "id": "did:example:5e0fe086-3dd7-4b9b-a25f-023a567951a4",
 "type": "Service",
 "name": "Dummy Bot",
 "publicKey": [{
   "id": "did:example:5e0fe086-3dd7-4b9b-a25f-023a567951a4#key-1",
   "owner": "did:example:5e0fe086-3dd7-4b9b-a25f-023a567951a4",
   "publicKeyPem": "-----BEGIN PUBLIC KEY-----\r\n..."}]}
```

Finally, here is the Cloud Storage service:

```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 "id": "did:example:0b36c784-f9f4-4c1e-b76c-d821a4b32741",
 "type": "Service",
 "name": "Cloud Storage Pro",
 "publicKey": [{
   "id": "did:example:0b36c784-f9f4-4c1e-b76c-d821a4b32741#key-1",
   "owner": "did:example:0b36c784-f9f4-4c1e-b76c-d821a4b32741",
   "publicKeyPem": "-----BEGIN PUBLIC KEY-----\r\n..."}]}
```

Alice's capability to store an object in the Cloud Store is encoded in a proclamation, which looks like this:

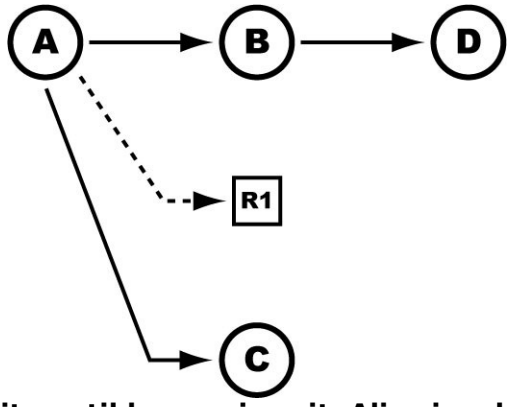
```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 "id": "did:example:0b36c7844941b61b-c763-4617-94de-cf5c539041f1",
 "type": "Proclamation",

 // The subject is who the capability operates on (in this case,
 // the Cloud Store object)
 "subject": "did:example:0b36c784-f9f4-4c1e-b76c-d821a4b32741",

 // We are granting access specifically to one of Alice's keys
 "grantedKey": "did:example:83f75926-51ba-4472-84ff-51f5e39ab9ab#key-1",

 // No caveats on this capability... Alice has full access
 "caveat": [],

 // Finally we sign this object with one of the CloudStorage's keys
 "signature": {
    "type": "RsaSignature2016",
    "created": "2016-02-08T16:02:20Z",
    "creator": "did:example:0b36c784-f9f4-4c1e-b76c-d821a4b32741#key-1",
    "signatureValue": "IOmA4R7TfhkYTYW8...CBMq2/gi25s="}}
```

Now Alice wants to share this capability to Bob, but with a couple of caveats (also known as an "attenuation"): Bob can only invoke the upload method, and can only upload 50 Megabyte files at a time.

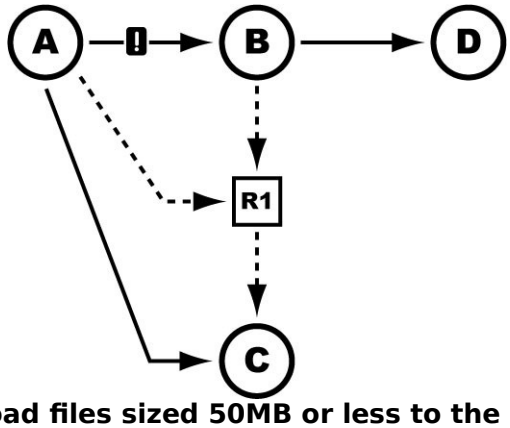
```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 "id": "did:example:f7412b9a-854b-47ab-806b-3ac736cc7cda",
 "type": "Proclamation",

 // This new attenuated proclamation points to the previous one
 "parent": "did:example:0b36c7844941b61b-c763-4617-94de-cf5c539041f1",

 // Now we grant access to one of Bob's keys
 "grantedKey": "did:example:ee568de7-2970-4925-ad09-c685ab367b66#key-1",

 // This proclamation *does* have caveats:
```

```
      "caveat": [
        // Only the UploadFile method is allowed...
        {"id": "did:example:f7412b9a-854b-47ab-806b-3ac736cc7cda#caveats/upload-
only",
         "type": "RestrictToMethod",
         "method": "UploadFile"},
        // ...and each upload can only be 50 Megabytes large.
        {"id": "did:example:f7412b9a-854b-47ab-806b-3ac736cc7cda#caveats/50-megs-
only",
         "type": "RestrictUploadSize",
         // file limit here is in bytes, so 50 MB
         "limit": 52428800}],

      // Finally we sign this object with Alice's key
      "signature": {
         "type": "RsaSignature2016",
         "created": "2016-02-08T16:02:20Z",
         "creator": "did:example:83f75926-51ba-4472-84ff-51f5e39ab9ab#key-1",
         "signatureValue": "..."}}
```

As this diagram demonstrates, Alice has created, and has access to, this attenuated capability.



Bob cannot use this capability until he receives it. Alice invokes her message sending capability between herself and Bob.

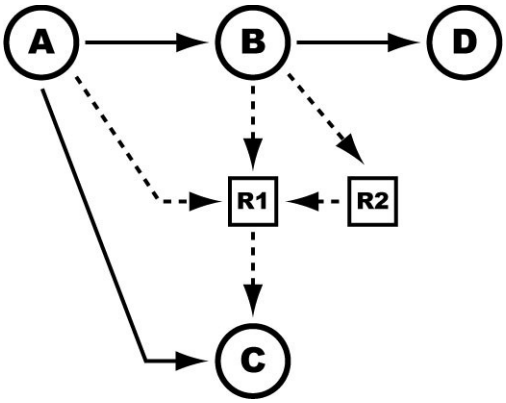


Now Bob has access to upload files sized 50MB or less to the Cloud Store. But he would prefer that Dummy Bot do uploads for him... well, for a month. He'll see how it goes.

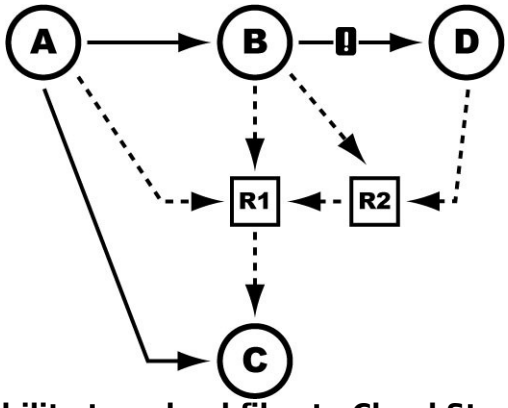
Luckily these capabilities are composable, and so Bob can create an attenuated capability out of the attenuated capability he already has!

```
      {"@context": ["https://example.org/did/v1",
                    "https://example.org/ocap/v1",
                    "http://schema.org"],
       "id": "did:example:d2c83c43-878a-4c01-984f-b2f57932ce5f",
       "type": "Proclamation",

       // Yet again, point up the chain...
       "parent": "did:example:f7412b9a-854b-47ab-806b-3ac736cc7cda",

       // Now we grant access to one of Dummy Bot's keys
       "grantedKey": "did:example:5e0fe086-3dd7-4b9b-a25f-023a567951a4#key-1",

       // We add a new caveat/attenuation: this one will expire 30 days
       // in the future
       "caveat": [
         {"id": "did:example:d2c83c43-878a-4c01-984f-b2f57932ce5f#caveats/expire-
time",
          "type": "ExpireTime",
          "date": "2017-09-23T20:21:34Z"}],

       // Finally we sign this object with Bob's key
       "signature": {
          "type": "RsaSignature2016",
          "created": "2016-02-08T17:12:28Z",
          "creator": "did:example:ee568de7-2970-4925-ad09-c685ab367b66#key-1",
          "signatureValue": "..."}}
```

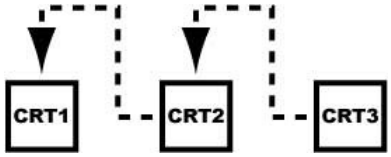
The capability graph now looks like this:



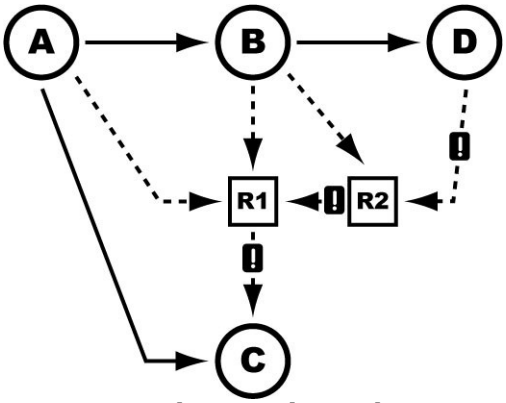
Bob invokes his message sending capability to send the new attenuated capability to Dummy Bot:



Now Dummy Bot has a capability to upload files to Cloud Store, but only files that are sized 50 megabytes or less, and only for the next month. These multiple caveats are possible because Dummy Bot is authorized on the final proclamation, and the proclamation "chains upward", including both the immediate restriction/caveat within R2 on time and also the restriction/caveat in R1 on space!



Soon Dummy Bot takes a picture and uploads it:



This is done through an `Invocation` on the proclamation, containing additional parameters in the body:

```
{"@context": ["https://example.org/did/v1",
              "https://example.org/ocap/v1",
              "http://schema.org"],
 "id": "did:example:2bdf6273-a52e-4cdf-991f-b5f000008829",
 "type": "Invocation",

 // Dummy Bot is invoking the proclamation it has,
 // but the whole chain will be checked for attenuation and
```

```
// verification of access
"proclamation": "did:example:d2c83c43-878a-4c01-984f-b2f57932ce5f",

// The method being used
"method": "UploadFile",

// The key Dummy Bot is using in this invocation
"usingKey": "did:example:5e0fe086-3dd7-4b9b-a25f-023a567951a4#key-1",

// Here's the base64 encoded file as part of the payload
"file": "nEOSQ7jbzBNg0Glup/FfeGDDzvLDvgEL36wcNpmbvKDgPy6+...",

// Finally we sign this object with Dummy Bot's key
"signature": {
   "type": "RsaSignature2016",
   "created": "2016-02-08T17:13:48Z",
   "creator": "did:example:5e0fe086-3dd7-4b9b-a25f-023a567951a4#key-1",
   "signatureValue": "..."}}
```

RELATED WORK

SPKI/SDSI

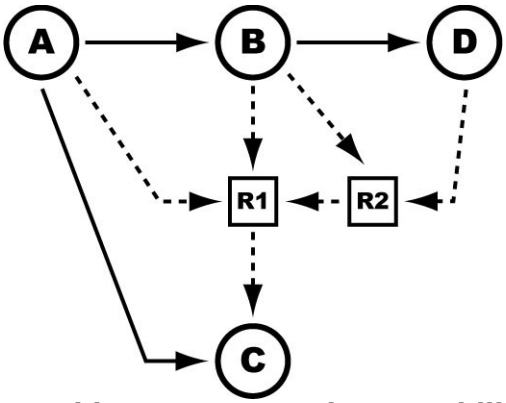
SPKI (and previously SDSI) is a key-management project that aimed to resolve many of the issues (including those surrounding centralization) that the X.509 infrastructure introduced and developed into over time. SPKI is almost but not quite an object capability system. (See From Capabilities To Financial Instruments and Capability Myths Demolished for more information.) SPKI uses "certificates" (akin to "proclamations"fn:1 here) to express authority, similar to what we are doing in this document, but did not exist in a linked data system as this proposal does. Importantly, SPKI's authority is a broader form of access control and for that reason carries some of the traditional problems of ACLs.

Macaroons

Macaroons are a credentials system that uphold most of the properties of capabilities. They also support delegation and attenuation (with some constraints as to who can attenuate) via a chain of signed messages, but there are some key differences.

The biggest advantage of Macaroons over our design is that messages are smaller (a desirable property!) because a simple HMAC is used for signing rather than public key cryptography. Macaroons are thus passed around as bearer instruments over secure channels. This leads to a tradeoff: macaroons are smaller in size than Linked Data Capabilities, but unlike Linked Data Capabilities, cannot be sent or invoked over an insecure channel. Unlike Linked Data Capabilities, macaroons cannot be stored on a blockchain or be publicly retrievable from the web.

One further difference is that while any entity that holds on to a macaroon may delegate that macaroon to any other entity, not all entities can attenuate macaroons. To see why, let us look at our final configuration between Alice, Bob, Dummy Bot, and Cloud Store:



In this configuration, Alice was able to attenuate her capability to Cloud Store before delegating to Bob without any specific permission to do so; Bob was likewise able to attenuate the attenuated capability he held before passing to Dummy Bot without any specific permission.

In Macaroons, Cloud Store and Alice must pre-arrange the shared key that Alice will use to attenuate the macaroon she holds before she can do so and successfully delegate to Bob (likewise for Bob to Dummy Bot). The reason for this is that in verifying HMAC signatures Cloud Store must check the macaroon's signatures against a key that Alice and Cloud Store must both have... Alice to sign it and Cloud Store to verify it. Even if Alice and Cloud Store had prearranged a shared key to be used for attenuating macaroons, if Bob and Cloud Store had not done so there would be no way for Bob to further attenuate the capability before passing to Dummy Bot. Bob may not prefer this to be the case since Bob wanted to only give Dummy Bot access for thirty days.

(Notably, the Macaroons paper contains a short but underspecified section that outlines how Macaroons could be used with public keys instead of HMAC-signed bearer instruments; the design described, while scantly detailed, sounds very similar to how Linked Data Capabilities work.)

Overall Macaroons and lds-ocaps are both reasonable systems with different tradeoffs. Implementers should be informed of these tradeoffs and make decisions accordingly.

Object Capability Programming Languages

Up until this point this paper has focused on different substrates on which to implement capabilities, which have all relied on some sort of shared vocabulary between entities in the system. Another way to build capabilities is to build them at the layer of a programming language. In addition to not requiring coordination on vocabulary from all entities in the system, this provides powerful compositional abilities which, as we will see, turn out to be highly desirable.

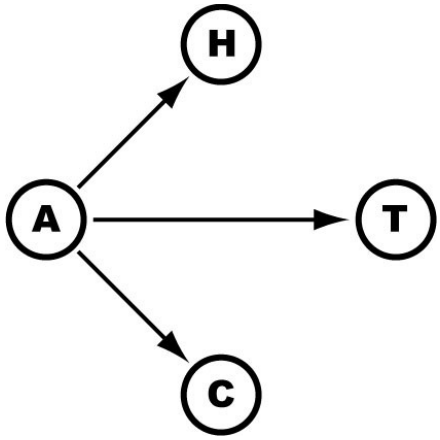
In the W7 Security Kernel, Jonathan Rees introduces an implementation of object capabilities on nothing other than a strict lexically scoped environment, enforced by the runtime of the system. The example language uses a cut-down variant of Scheme, though it could be implemented in any language that provides the same strict lexical scoping properties in a carefully bounded initial environment. (This is the general mechanism for implementing capabilities at a programming language level.) The paper demonstrates all the same properties of capabilities demonstrated here: delegation, attenuation, and so on.

However, there is one thing that is possible in W7 (and other similar systems) that is not possible in any of the other systems discussed in this paper, including the lds-ocaps system herein proposed.

This is attenuation by composition in an enclosed environment. To see what this means and why it is desirable, consider this example.

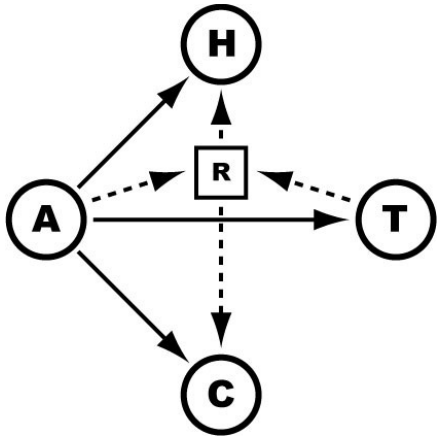
We have the following initial state:

- A: Alice
- C: Cloud Store
- H: Home Directory
- T: Timer Service



(A)lice keeps her data in (H)ome Directory. She would like to back it up to (C)loud Service, but she is afraid she will forget to back up regularly, so she would like to grant a capability to (T)imer Service to run the backup for her. However, she would prefer that Timer Service not have access to actually read any of the contents of her data on Home Directory, and she does not want Timer Service to be able to write just anything to Cloud Store, only backups. Effectively she would like to send Timer Service a new capability that *composes together* reading from Home Directory and writing to Cloud Store without giving access to either independently.

Here R represents the restricted-through-composition capability:



In Rees' W7 / lambda-calculus-ocap system, this could be represented as:

```
;; Run in A's environment
(timer-run-every            ; T
  (lambda ()                ; R
    (write-cloud-image      ; C
      (get-homedir-image))) ; H
  (* 60 60 24 7)) ; run every 604800 seconds, or once a week
```

The advantage here is that the runtime is able to enclose the capabilities and handle the composition of passing the returned value of one of the enclosed capabilities to the other, without exposing either individually, outside of the enclosure.

It does not appear that lds-ocaps can do the same thing. Here is a highly cut down invocation that attempts to embed the capabilities, for the sake of demonstration:

```
{"type": "Invocation",
 "usingKey": <alice-key-1>,
 "method": "RunEvery",
 "proclamation": <cert-id>,
 "secs": 30,
 /* But we would also need to clearly express how to combine these */
 "runTheseCombinedSomehow": [
   <alice-grants-timer-capability-to-homedir>,
   <alice-grants-timer-capability-to-cloudstore>],
 "signature": <signed-by-alice-key-1>}
```

There are two troubles here: making it clear how Timer Service is supposed to compose these capabilities in runTheseCombinedSomehow is not obvious, and even worse, there is nothing preventing Timer Service from running these individually since they are not properly enclosed, unlike in the W7/Scheme example.

The majority of needs for a capability system are likely served by attenuation and delegation on their own. Nonetheless, full composability within a capability's enclosure, as explored above, is still a desirable property for the systems that can provide it.

CapCert

CapCert is a (currently unimplemented) plan for a proclamation/certificate chain-based structure that looks a lot like the system discussed in this paper with one interesting change: real programs may be embedded *in* the proclamations. This approach bridges the gap between the proclamation-chain approach described in this paper and the object-capability programming languages described in the previous section; proclamations can be shared over insecure channels while also removing some need for shared vocabulary on both ends. Even more excitingly, the kinds of composition we do not have but would like to have would be possible, such as the example given above of Alice allowing a Timer Service to back up her Home Directory to Cloud Store, without giving Timer Service access to either independently.

It would be possible to build such a system with Linked Data Capabilities by embedding an object capability programming language (with proper constraints on space and time for safety as well). This is a significant topic worth its own future paper.

Capabilities on Blockchains

Finally, one piece of related work that we have not addressed but would like to address in a future paper is enabling capabilities on blockchains. Motivating examples include attacks against Etherium smart contracts that would not have occurred in an object capabilities environment.

The examples provided here demonstrate capabilities that may exist in environments where the only secrets that must be kept are the private keys of entities participating in the system. We would like objects committed to a blockchain to be able to express capabilities despite not being able to hold secrets on the blockchain itself. This is also a significant topic worth its own future paper.

CONCLUSIONS

Linked Data Systems are powerful ways to build collaborative, expressive systems. Today we are seeing Linked Data Systems crossing not only the traditional web but even into systems like distributed ledger technologies and so on. Unfortunately, security is frequently difficult on Linked Data Systems.

For example, SoLiD directly uses and ActivityPub indirectly implies Access Control Lists. Unfortunately these are are known to create problems in systems, particularly:

- excess authority leading to needless vulnerability
- ambient authority leading to confused deputy problems
- lack of composability (including attenuation)

We can avoid these risks by using an object capability system such as the one described above. Even more exciting, by combining this system with DIDs we can build a fully decentralized object capability system for the web that is safe to use.

ADDITIONAL CREDITS

Authors: Christopher Lemmer Webber & Mark S. Miller

About Rebooting the Web of Trust

This paper was produced as part of the Rebooting the Web of Trust V design workshop. On October 3rd through October 5th, 2017, over 50 tech visionaries came together in Cambridge, Massachusetts to talk about the future of decentralized trust on the internet with the goal of writing 3-5 white papers and specs. This is one of them.

Preliminary Workshop Sponsors List: BigChainDB, Blockchain Lab, Digital Contract Design, IDEO, IPFS, Protocol Labs, Toni Lane Casserly

Workshop Producer: Christopher Allen

Workshop Facilitators: Christopher Allen, with additional paper editorial & layout by Shannon Appelcline.

What's Next?

The design workshop and this paper are just starting points for Rebooting the Web of Trust. If you have any comments, thoughts, or expansions on this paper, please post them to our GitHub issues page:

> https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust-fall2017/issues

The next Rebooting the Web of Trust design workshop is scheduled for early 2018 on the west coast of the USA. If you'd like to be involved or would like to help sponsor these events, email:

> ChristopherA@LifeWithAlacrity.com
