Analysing Object-Capability Patterns With Murϕ

Deian Stefan

Abstract

Object Capability (OCap) patterns provide an alternative, more secure, approach to building systems following the principle of least authority. Despite the increasing popularity and application of OCap patterns, little effort has been put into the analysis of these patterns. We analyze several OCap patterns using the Murϕ verification tool. Our results confirm previous vulnerabilities found in several OCap patterns when used in an operating systems context. We verify an attack on the sealer/unsealer pattern using the Joe-E with Java threads. Finally, we propose alternative patterns that our model checking did not find to be vulnerable. We including code that further demonstrates their usability.

1 Introduction

A wide number of system exploits and exposures take advantage of the ample vulnerabilities[1], e.g. buffer overruns, found in large programs. Even security-concious programmers are susceptible to writing exploitable code, as many abstractions we are accustomed to are inherently flawed. Consider, for example, the widely-used POSIX API. In a POSIX environment, a simple logging application, `aLogger`, running on behalf of user `alice` and writing messages to `/var/log/alice.log`, is actually given the full authority of user `alice`. This *ambient authority* is assumed by many POSIX system calls. For example, the system call

```
open(const char *pathname, int flags);
```

can be used by an application to open any file the invoking user can open. Considering the logging example, even if `aLogger` is *honest* in only opening `/var/log/alice.log`, an exploit modifying the path name can result in `aLogger` overwriting sensitive files, including, for example, `alice`'s private cryptographic keys. This is possible since `aLogger` has ambient authority and can overwrite whatever file `alice` can. Worse yet, applications are widely distributed in binary form and verifying that an application is honest is usually not possible. Therefore, users cannot safely execute such applications without extensive sanboxing or creating different user account for each application. These methods are, however, cumbersome and usually cripple the application—hence, users usually run applications with the hope that they are not malicious.

A more secure approach, following the *Principle of Least Authority* (POLA) can be used to run `aLogger` by first opening `/var/log/alice.log` and providing `aLogger` the file descriptor. Informally, POLA states that an application should be given enough authority to do its job, but nothing more. Although the modified `aLogger` will, in actuality, be running with ambient authority, if all `open` system calls are disallowed, a POLA-like setting can be created and `aLogger` will not have the authority write to anything other than the file descriptor provided. Of course, many applications using POSIX (and many other APIs) rely on ambient authority, a side effect of insecure APIs and patterns programmers have gotten accustomed with.

Furthermore, our current insecure approaches to building systems are not usually "patchable", which highlights the need for alternative methods of composing secure systems (without trading off expressiveness and functionality). One such alternative is the Object Capability (OCap) model [9, 8, 6].

[1] See `http://cve.mitre.org/cve/cve.html`

OCap model extends the object model by disallowing things common in object model implementations, including mutable static state, forged pointers and ability of an object to access another's private state [8]. The OCap model is increasingly becoming more popular, yet little effort outside [11, 13, 12, 4, 17] has been put into formal analysing or modeling of its patterns. In this work, we extend Murray's work [11] by analysing several OCap patterns using Murϕ [1, 10], and present new patterns that address the limitations of the analyzed-patterns in a distributed, or operating systems, environment.

2 OCap model and patterns

In the OCap model there is no distinction between a subject (e.g. Alice) and and an object (e.g. a file, or class instance). Rather, everything is considered an object and all communication is accomplished by sending messages on references. Following [8], an object can be a *primitive*, such as the literal `2`, or an *instance*[2] which is a combination of code and state. An object's state may include references to primitive objects or other instances. The latter, a reference to an instance, is called a *capability*. As already mentioned, all communication in an OCap system is done by sending messages on references, which implies that, in order for an object to send a message to another, it must have a capability to it. Of course, these messages may include capabilities, and so the reference graph, which is also the access graph, changes as capabilities are acquired and dropped. In an OCap system an object may come to posses a capability only through the following methods:

- *Initial conditions*.
- *Parenthood*. When object A creates another object B, it is the only object in the system with the capability to B.
- *Endowment*. If object A has a capability to object C, then it can create another object B such that B is already 'endowed' with a capability to C.
- *Introduction*. If object A has capability to objects B and C, then A can give B (resp C) capability to C (resp B) by sending it a message containing the capability.

[2] A process is considered instance of a program, similar to an object being an instance of a class.

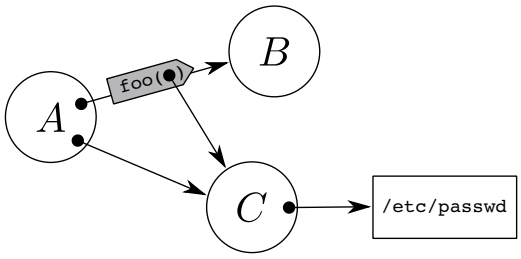

Figure 1: Connectivity by introduction: A send B message `foo` with capability to C.

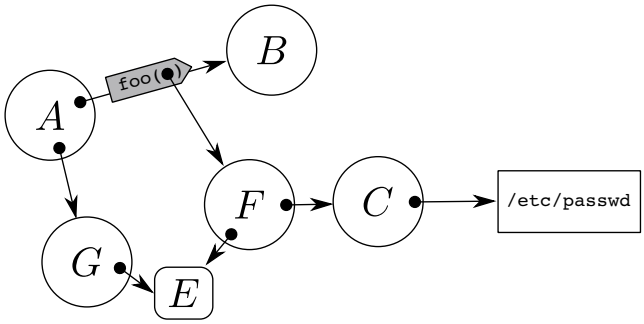

Figure 2: Selective revocation: A send B message `foo` with capability to F, and, indirectly, revocable access to C.

A direct consequence of these rules is that "only connectivity begets connectivity" and so a message from one subgraphs cannot be sent to another if they are disjoint. This is particularly important in building secure systems because at each 'snapshot' of the dynamic graph, it is directly clear who has access to what and the connectivity rules state how the graph may change. Consider, for example, the graph where A has capability to B and C. The latter, C, has a read/write capability to file `/etc/passwd`. The only way B can gain *any* access to `/etc/passwd` is through A introducing C (or some forwarder) to B, as shown in Figure 1.

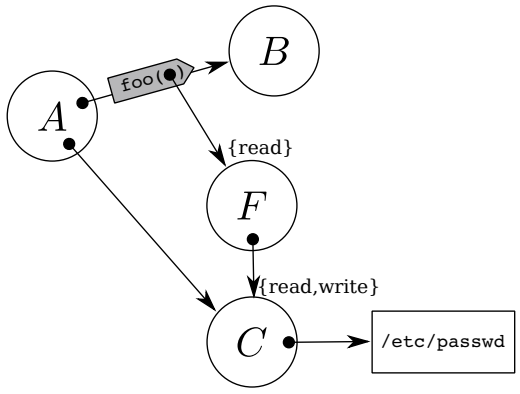

Figure 3: Attenuated forwarders: A send B message `foo` with capability to F, and, indirectly, read-only access to C.

An immediate concern with capabilities, however, is *revocation*. Because a capability is simply a reference to an object, it is not clear how one can revoke the capabilities it grants. Simply, a capability cannot be revoked. However, using a revocation *pattern* an object can revoke the access a granted capability has. Though other patterns are conceivable, a common revocation pattern is Redell's caretaker pattern [14, 8]. Using the caretaker pattern A can give B revocable access to C as follows. First, A creates three objects:

- A mutable boolean (enable) slot E.
- A gate G, with capability to E. Upon receiving message `toggle`, G toggles E's value.
- A *forwarder* F, with capability to E and C. Upon receiving message m, F checks the value of E and only if it is true, it proceeds to forward m to C.

Now, instead of giving B capability to C, A gives B capability to F, as shown in Figure 2. While E remains true B can send any messages to C, as if F is not in the reference path. However, A can revoke B's access to C at any time, simply by sending G the message `toggle`.

A concern of equal interest to revocation is *attenuation*. Specifically, suppose A wants to give B a capability to C, however, restricting B's privilege to a certain message, e.g., `read`. To do so, as in the caretaker pattern, A creates a forwarder, F. In this case, F only accepts the message `read`, as shown in Figure 3. Of course, this read-only forwarder pattern can be combined with the caretaker pattern to grant a revocable, read-only capability.

Note that this pattern only restricts B's privilege to C through F. If C responds to B's `read` message with a capability to itself then B's authority will entail write access to `/etc/passwd` as well. Thus, if C cannot be trusted to respect the read-only capability, A must use a membrane forwarder, instead of a read-only forwarder. A membrane forwarder [8], wraps every capability in either direction and thus if C returns B a raw capability to itself, the membrane will wrap it, effectively making it read-only. As in the case of the read-only forwarder, the membrane pattern can be combined with the caretaker pattern to grant a revocable, *transitive* read-only capability.

The final pattern we consider, is that of right amplification using sealer/unsealer pairs [6, 16]. The basic premise of sealer/unsealer pairs is similar to that of public-/private-key pair: an object A may use a sealer to "seal" another object \$ which C can unseal only if it has the corresponding unsealer. Though some OCap systems, such as E, support sealer/unsealer pairs as a primitive, other implement it using a pattern. Following [16], the sealer/unsealer pattern consists of

- A mutable shared slot S.
- A sealer with capability to S, that, upon receiving the message `seal` with argument \$, creates and returns a box B (with endowed capability to S). Box B, when invoked, writes \$ into slot S.
- An unsealer U with capability to S, that, upon receiving the message `unseal` with argument B', clears S, and invokes B'. After invoking B' it returns the slot contents, if any.

Note that for the unsealer to return \$, it must invoke box B' such that $B = B'$. As an example use, consider Figure 4. In this case object A can send C a capability to \$ through the curious object D.

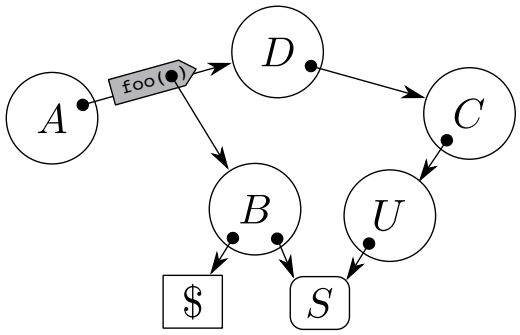

Figure 4: Sealer/unsealer pattern: A sends C a capability to \$ through D. Using a sealer, A sends D a capability to box B. Upon receipt of B, C can use unsealer U and box B (which writes \$ to S) to retrieve \$

.

3 Modeling OCap patterns in Murϕ

Using the Murϕ verification tool we model several aspects of the OCap model, including the patterns discussed in the previous section. We, however, note that for each modeled pattern we initialize our system with a small access graph consisting of several interconnected objects. Since we do not model object creation, the dynamics of the access graph only changes based on the rule of introduction.

As in [11], we are interested in modeling OCap patterns in the context of programming languages (PL) and distributed/operating systems (D/OS). As a number of capability systems in these domains already exist [2, 15, 3, 18, 5] , we believe that our analysis is useful towards understanding the security properties of such OCap patterns when used in complex systems. More importantly, we believe a recommendation on how to securely implement these patterns in both contexts is essential. As we show in Section 4 and Murray previously showed in [11], a direct implementation of several of the patterns will violate the security properties they are designed to uphold.

Each object in the system is modeled by a single type `Object`, which includes the object's state, capability list (c-list), and behavior description (i.e. whether it is a forwarder, a gate, a membrane, an unsealer, or 'plain'). This closely corresonds to the definition of an instance consisting of state (c-list) and code (behavior). Furthermore, we note that a capability in an object's c-list can be a *raw* capability or a *wrapped*, i.e. attenuated, capability.

We model object communication using a simple network whose buffer length is 1 in the case of PL, and > 1 in the D/OS setting. Each inter-object-communication message (IOCM) contains several standard fields, e.g. source, destination, message, message arguments, etc. Note that an IOCM is used to send messages, specifically `call, return, grant, revoke` and `unseal`, between objects. Each IOCM additionally tracks the message path (i.e. intermediate objects a message has passed through), a time stamp of when the message was placed on the network, and whether or not the message is wrapped. The architecture we model is a typical call-return blocking architecture and thus only objects in an idle state may invoke a capability from their c-list, at which point they will go in a blocking state, awaiting a `return` message.

Given an overview of our object and communication model we now detail the specific patterns me modeled, their security properties, and our approach to verifying the invariance of these properties. We use $\mathcal{S}(m) \xrightarrow{m(a)} \mathcal{D}(m)$ to denote a message m with argument a being sent from source $\mathcal{S}(m)$ to destination $\mathcal{D}(m)$. A $\xrightarrow{m(a)}{}^{+}$ indicates a series of message, the last of which is m, i.e. transitive (but not reflexive) relation.

- **Membrane forwarder pattern**: The property we expect the membrane pattern to uphold is *transitive attenuation*. Recall that each IOCM tracks the message path, and whether a message is raw or wrapped. If object O_j receives a raw message from O_i over path π, we say the paths from O_k to O_j, for $k \in \pi$ are *tainted*. Each object state contains a taint list T such that $T(O_j, O_k)$ is true if the path from O_j to O_k is tainted. Let $\mathcal{M}(O_i)$ hold true if O_i is a membrane object and false otherwise. We express the pattern property as invariant:

$$\forall O_i, O_j . \mathcal{M}(O_i) \wedge \neg T(O_i, O_j)$$

- **Revocable membrane forwarder pattern**: Although our implementation can model a simpler non-membrane revocable forwarder pattern, the membrane version 'subsumes' the former by further incorporating the above invariant. Following Figure 2, we expect *the forwarder F to only forward messages when the enable flag E is true*. Our model implements a gate that, in addition to toggling the enable flag E, also records the time[3] when the value is changed, $t(E)$. Moreover, recall that each message m has a corresponding time stamp $t(m)$ indicating when it was sent. Though our model invariant is expressed generally, the pattern property for the single-revoker of Figure 2 is given as invariant:

$$\nexists m.\mathcal{S}(m) = F \wedge \neg E \wedge t(E) < t(m)$$

- **Right amplification pattern**: For the sealer/unsealer pair pattern we expect *the unsealer to not be able to gain access to the slot contents, unless it has the (correct) box returned by the corresponding sealer*. Again, for clarity, we express this property for the simpler setup, Figure 4, as invariant:

$$\nexists B'.B' \neq B \wedge C \xrightarrow{\texttt{unseal}(B')} U \xrightarrow{\texttt{return}(\$)} {}^{+}C$$

4 Analysis and discussion

Given the description of our Murϕ model and pattern invariants, for each pattern we ran the verifier only modifying the starting state to adjust for the PL or D/OS setup and the initial access graph. We summarize our results below.

- **Membrane forwarder pattern**: As expected, we did not find any system property violations in either the PL or D/OS setting(even varying the number of concurrent messages in the latter case). Although Murϕ found no vulnerabilities for this pattern, we stress that this pattern holds (up to our model) for OCap systems; implementing membrane-like patterns in other capability system does not guarantee transitive attenuation as suggested in [7]. In the Appendix A we show a violation of a transitive read-only property for the file system TahoeLAFS [19].

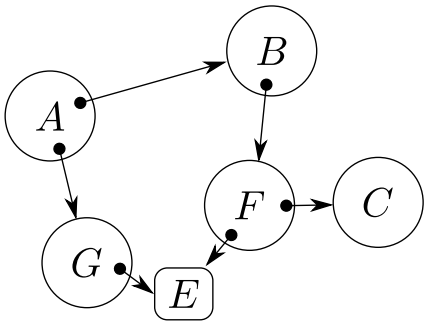

Figure 5: Revocable membrane forwarder: F is a membrane forwarder to C, G is a gate, and E is the enable slot used by the two.

- **Revocable membrane forwarder pattern**: In the PL setting, as in the previous case we found no violations. However, as in [11], we founds that the revocable membrane pattern in a D/OS setting has a time-of-check-to-time-of-use vulnerability. Consider the access graph of Figure 5, where A granted B attenuated access to C. Starting with E enabled, and using `isEnabled` for the message used by the forwarder to check status (which in our model is implicit), Murϕ found the following sequence of message calls to violate the invariant:

1. $B \xrightarrow{\texttt{call}(\cdot)} F$
2. $F \xrightarrow{\texttt{isEnabled}()} E$
3. $E \xrightarrow{\texttt{return}(true)} F$
4. $A \xrightarrow{\texttt{toggle}()} G \rightarrow E$
5. $E \rightarrow G \xrightarrow{\texttt{return}()} A$
6. $F \xrightarrow{\texttt{call}(\cdot)} C$

[3] We model time as a simple counter. Any instance a message is place on the network or a revocation enable slot value is changed, the global counter is incremented. Because our models are very small, this does not result in overflows or other modeling anomalies.

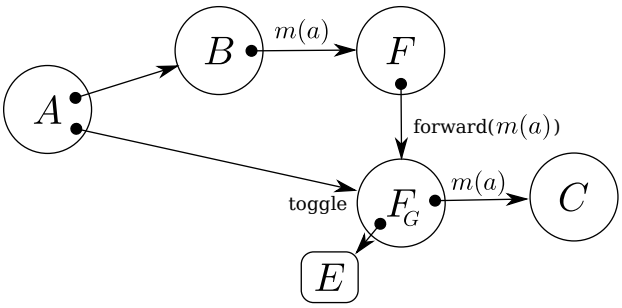

Figure 6: Proposed revocable forwarder: F_G is a forwarder gate to C, F is a wrapping forwarder, and E is the enable slot used by the F_G.

Note that F checked status in step 3, but did not use the value returned by E until step 6, at which point E was toggled. Addressing this vulnerability, we propose a slightly modified revocation pattern, shown in Figure 6. For simplicity we present a non-membrane pattern, extending it to a membrane pattern can be done as in [8]. This pattern consists of a wrapping forwarder (note, in this case 'wrap' is not used to interchangeably with membrane) F, which takes any message $m(a)$ and sends it as an argument to the message `forward` to the forwarding gate F_G. F_G handles two messages, `toggle` which is used to revoke access, and `forward` which is handled by forwarding the inner message to C. Figures 7 and 8, show the code for the original and our proposed revocation patterns, respectively. Note that although we provide E code, we do not claim that the vulnerability is applicable to E. After running Murϕ for 13,720,000 states using the proposed pattern, no invariant failed.

- **Right amplification pattern**: Similar to the above and as in [11], in the PL setting, we found no violations. However, we found that the sealer/unsealer pair pattern in a D/OS setting allows for the unsealer to gain access to the slot written by the box without having the correct box. Consider the initial dynamic access graph shown in Figure 9, where B' is a box object that C created by sealing an arbitrary object. Murϕ

```
def makeCaretaker(target) {
  var enabled := true
  def caretaker {
    match [verb, args] {
      if (enabled) {
        E.call(target, verb, args)
      } else {
        throw("diabled")
      }
    }
  }
  def gate {
    to toggle()  { enabled := !enabled }
  }
  return [caretaker, gate]
}
```

Figure 7: Original revocation pattern in E, based on [8].

```
def forwardingGate(target) {
  var enabled := true
  def gate {
    to toggle()  { enabled := !enabled }
    to forward(verb, args) {
      if (enabled) {
        E.call(target, verb, args)
      } else {
        throw("diabled")
      }
    }
  }
  return gate
}

def makeCaretaker(target) {
  var gate := forwardingGate(target)
  def caretaker {
    match [verb, args] {
      gate.forward(verb, args)
    }
  }
  return [caretaker, gate]
}
```

Figure 8: Proposed revocation pattern in E.

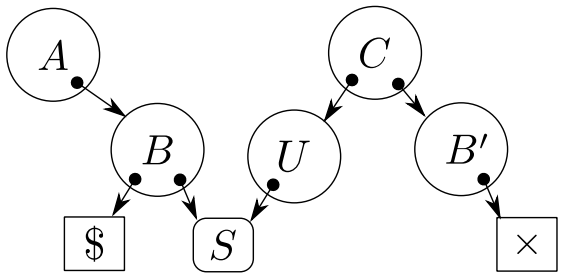

Figure 9: Sealer/unsealer pattern initial dynamic graph. A has a capability to box B which has capabity to \$ and slot S, that is common to the unsealer U. C has capability to box B'.

found the following sequence of message calls, equivalent to the attack in [11], to violate the invariant:

1. $C \xrightarrow{\texttt{unseal}(B')} U$
2. $U \xrightarrow{\texttt{clear}()} S$
3. $S \xrightarrow{\texttt{return}()} U$
4. $U \xrightarrow{\texttt{call}()} B'$
5. $B' \xrightarrow{\texttt{return}()} U$
6. $A \xrightarrow{\texttt{call}()} B$
7. $B \xrightarrow{\texttt{return}()} A$
8. $U \xrightarrow{\texttt{read}()} S$
9. $S \xrightarrow{\texttt{return}(\$)} U$
10. $U \xrightarrow{\texttt{return}(\$)} C$

We note that our model does not explicitly model the interaction with the slot as we have shown; specifically, `clear` and `read` are inlined and are not messages being sent to an object. The basic observation of this vulnerability is that C can call the unsealer with any box (e.g. B'), and after the unsealer has cleared the slot, if the correct box B is invoked the slot is filled with \$, which the unsealer then proceeds to read and return back to the caller (having assumed it was B' that filled the slot).

```
public class SealerUnsealer {
  private Object shared;

  public Box Seal(Object o){
    return new Box(o);
  }

  public Object Unseal(Box box) {
    shared=null; //clear
    box.share();
    return shared;

  }

  public class Box {
    private final Object box_shared;
    public Box(Object o) {
      box_shared = o;
    }
    public void share() {
     shared = box_shared;
    }
  }
}
```

Figure 10: Joe-E sealer/unsealer pattern, based on the E implementation of [16].

We implemented this attack in the JoE-E language with two Java threads behaving as A and C. Figure 10 shows the JoE-E implemented pattern used in our tests. As in the case of the revocable forwarder, we propose an alternative pattern that we verified with Murϕ to satisfy the invariant (up to the model). Figure 11 show the proposed pattern, in Joe-E, that differs from the original pattern as follows. Rather than implementing the box to simply write the capability to object \$ in the slot, the box should also write the capabilty to itself. The unsealer, invoked with B' clears the slot, invokes B', and then reads the contents of the slot. If the box-capabilty is the same as B', the box it invoked, then the unsealer returns the object placed in the slot, i.e. \$. Otherwise it fails (in our case, silently).

5 Conclusion

We modeled several Object Capability patterns using the Murϕ verification tool. Our results confirm

```
public class SealerUnsealer {
  private Object shared;
  private Object modifier;

  public Box Seal(Object o){
    return new Box(o);
  }

  public Object Unseal(Box box) {
    shared=null;
    box.share();
    if(!modifier.equals(box)) return null;
    return shared;

  }

  public class Box {
    private final Object box_shared;
    public Box(Object o) {
      box_shared = o;
    }
    public void share() {
     shared = box_shared;
     modifier = this;
    }
  }
}
```

Figure 11: Joe-E sealer/unsealer proposed pattern.

Murray's previous work in finding a vulnerability in the revocable forwarder pattern and sealer/unsealer pattern when used in a distributed or operating systems context. We further implement the latter attack in Joe-E with Java threads, and finally propose alternative patterns that address these vulnerabilities.

References

[1] D. Dill. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393. Springer-Verlag, 1996.

[2] N. Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[4] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*, pages 125–140. IEEE, 2010.

[5] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *17th Network & Distributed System Security Symposium*, 2010.

[6] M. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *Financial Cryptography*, pages 349–378. Springer, 2001.

[7] M. Miller, K. Yee, J. Shapiro, et al. Capability myths demolished. Technical report, Johns Hopkins University, Tech. Rep, 2003.

[8] 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.

[9] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003.

[10] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Mur/spl phi. *sp*, page 0141, 1997.

[11] T. Murray. Analysing object-capability security. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

[12] T. Murray. *Analysing the security properties of object-capability patterns*. PhD thesis, University of Oxford, 2010.

[13] T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. *Formal Aspects in Security and Trust*, pages 81–95, 2010.

[14] D. Redell and D. Redell. Naming and protection in extendable operating systems. 1974.

[15] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawa Island, SC, December 1999. ACM.

[16] M. Siegler. A picturebook of secure cooperation. Presentation, 2004. erights.org/talks/efun/SecurityPictureBook.pdf.

[17] A. Spiessens. *Patterns of safe collaboration*. PhD thesis, Université catholique de Louvain, February 2007.

[18] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *Creating, Connecting and Collaborating through Computing, 2007. C5'07. The Fifth International Conference on*, pages 163–169. IEEE.

[19] 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*, pages 21–26. ACM, 2008.

A TahoeLAFS's transitive read-only

The Tahoe Least Authority File System (TahoeLAFS) [19] is a file system that uses capabilities for access control. Specifically, TahoeLAFS uses a URI consisting of random string to uniquely identify a file/directory; following the capability model, a capability does not separate authority and designation. However, their their model is not an OCap mode, and thus, they cannot guarantee

> ... the property of transitive read-only – users who have read-write access to the directory can get a read-write-cap to a child, but users who have read-only access to the directory can get only a read-only-cap to a child. It is our intuition that this property would be a good primitive for users to build on, and patterns like this are common in the capabilities community ... [19]

Consider the simple setup in which A has read-write access to directory D, and B has only read aces. Hence, B *should* only have transitive read-only access. An attack is directly apparent given that TahoeLAFS does not have a method of distinguishing between data and capabilities. Specifically, A can create a file F in D consisting of the read-write capability to D (or any subdirectory). B, using just the read-only reads file F at which point it has acquired read-write authority to D—highlighting a trivial violation, which we confirmed to work with TahoeLAFS public test grid. To be fair, the authors do claim "transitive" and not "transitive and reflexive", however A could easily have written the read-write capability of any sub directory or file.