Delegating Responsibility in Digital Systems: Horton's "Who Done It?"

Mark S. Miller
Hewlett-Packard Labs

Jed Donnelley
NERSC, LBNL

Alan H. Karp
Hewlett-Packard Labs

May 26, 2007

Programs do good things, but also do bad,
making software security more than a fad.
The authority of programs, we do need to tame.
But bad things still happen. Who do we blame?

From the very beginnings of access control:
Should we be safe by construction,
or should we patrol?
Horton shows how, in an elegant way,
we can simply do both, and so save the day.

with apologies to Dr. Seuss

1 Introduction

There are two approaches to protect users from the harm programs can cause, *proactive control* and *reactive control*. Proactive control should help prevent bad things from happening, or limit the damage when they do. But when repeated patterns of abuse occur, we need some workable notion of "who" to blame, so we can reactively suspend the access of the responsible party. For example, granting read-only access to a wiki proactively prevents that party from modifying the contents. Knowing who posted abuse allows reactively suspending that party's write access.

In the 1960's and 1970's the dominant access control paradigms were Access Control Lists (ACLs) and capabilities. ACL systems consider a program to be acting on behalf of its "user". Access is allowed or disallowed by checking whether this operation on this resource is permitted for this user. A capability—like an object-reference in a memory-safe language—is a communicable and unforgeable token used both to designate some object and to permit access to that object. Because the term "capabilities" has since been used for many alternative access control rules[10], we now refer to the original pure model[1] as *object-capabilities* or *ocaps* for short.

By allowing the controlled delegation of authority, ocap systems support proactive control well. The invoker of an object normally passes as arguments just those objects (capabilities) that the receiver needs to carry out that request. A user can likewise delegate to an application just that portion of the user's authority the application needs[21], limiting damage should it be corrupted by a virus.

Because ocaps operate on an anonymous "bearer right" basis, they seem to make reactive control impossible. Indeed, although many historical criticisms of ocaps have since been refuted[10, 15, 9, 16], a remaining unrefuted criticism is that they cannot record who to blame for which action[5].

Only ACLs currently support reactive control. By tagging all actions with the identity of the user they allegedly serve, they can log who to blame, and whose access to suspend. But ACLs are weak at proactive control. Solitaire runs with all its user's privileges. If it runs amok, it could do its user great harm.

The desire for reactive control has led some to forego the benefits of ocaps. One answer would be to try to mix elements of the two paradigms in one security architecture[6, 3]. Another is to emulate some of the virtues of one paradigm as a pattern built on the other. For example, Polaris[19] uses lessons learned

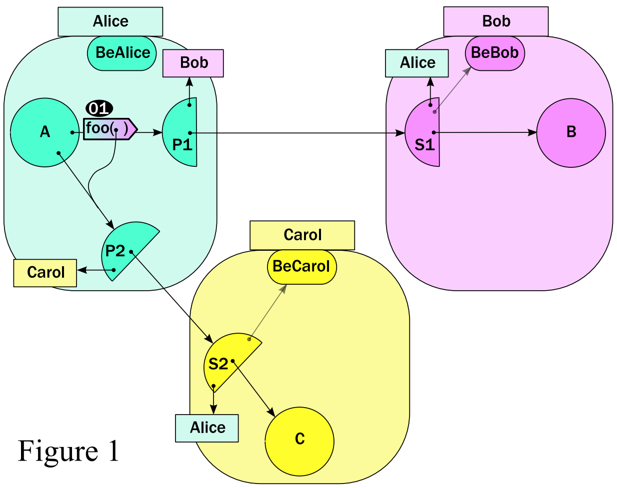

Figure 1

from ocaps to limit the authority of ACL-based applications on Windows, as Plash[14] does on Unix, without modifying either these applications or their underlying ACL OS.

In this paper, we show how to attribute actions in an ocap system. As a tribute to Dr. Seuss[4], we call our protocol Horton (*H*igher-*O*rder *R*esponsibility *T*racking of *O*bjects in *N*etworks). Horton can be interposed between existing ocap-based application objects, without modifying either these objects or their underlying ocap foundations. Horton supports identity-based tracking and control for delegating responsibility with authority. Horton thereby refutes this criticism of the ocap paradigm.

2 The Horton Protocol

Every protocol which builds some sort of secure relationship between its players must face two issues: 1) the base case, building an initial secure relationship between players not yet connected by this protocol, and 2) the inductive case, in which a new secure relationship is bootstrapped from earlier assumed-secure relationships. Horton contributes nothing to the issues of initial connectivity, so this paper only treats the inductive case. Starting from the situation in Figure 1 and the assumption that Alice, Bob, and Carol are mutually suspicious, our goal is to show how they arrive at the situation shown in Figure 3.

As with object computation, ocap references are conveyed as arguments in messages from a sender to a receiver. Here, we examine a scenario in which sending object A executes `b.foo(c)`, intending to send the message "foo" to receiver B with a reference to object C as an argument (Figure 1, ❶).

The circles A, B, C in the figures are application-level objects that need not be aware of Horton. The boxes represent messages in transit. Other shapes depict the Horton-aware objects each party uses to adapt their app-objects to the Horton protocol. When a *proxy* (an outgoing half circle such as Alice's P1) receives an app-level message, it encodes and sends it (Figure 2, ⓭) in a `deliver` message to its corresponding *stub* (an incoming half circle such as Bob's S1). The stub decodes it into an app-level message which it delivers to its target object (Figure 3, ㉙). Read the PDF online to see the figures "animate" by flipping pages.

For Horton to be transparent, the message delivered to B in step (29) must have the same app-level meaning as the message sent by A in step (01). To complete the induction, the relationship represented by the new B→P3→S3→C path must have the security we need, *assuming* that the A→P1→S1→B and A→P2→S2→C paths already have this security.

To support reactive security, we need to attribute actions to responsible identities. Cryptographic protocols often represent an electronic identity as a key pair. For example, a public encryption key identifies whoever knows the corresponding private decryption key. Put another way, knowledge of a decryption key provides the ability to *be* (or *speak for*[8]) the entity identified by the corresponding encryption key.

In ocap systems, the sealer/unsealer pattern[11] provides a similar logic. Rectangles such as the one labeled "Alice" represent Who objects, providing a `seal(contents)` operation, returning an opaque *box* encapsulating the contents. All such rectangles with the same label are references to the same Who object. The corresponding BeAlice object provides the authority to *be* the entity identified by Alice's Who object. BeAlice provides an `unseal(box)` operation that returns the contents of a box sealed by Alice's Who. The large rounded rectangles and colors ag-

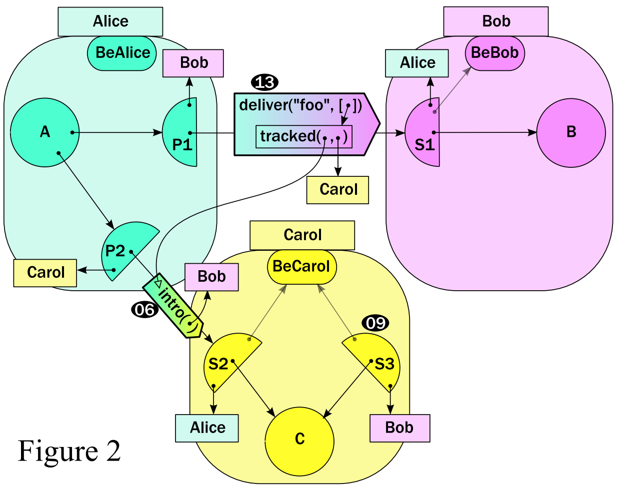

Figure 2

gregate all objects we assume to behave according to the intentions of a given Who.

Complete Horton implementations in the Java and E programming languages are available from `erights.org/download/horton/`. For expository purposes, this paper uses E to show just the Horton code needed for the illustrated case: sending a proxy reference as the single argument of a message with no return result. The line numbers on the code show the order of execution taken by our example. Mostly, this simplified code uses just the simple sequential five-minute subset of E explained in [9, Ch6: A Taste of E]. We also use some reflection which we explain as needed.

The code shown here implements a policy-free default behavior for participating in the Horton protocol. Each player expresses Horton-level policy—such as identity-based logging and revocation—by overriding these defaults.

When the `foo` message arrives at proxy P1, it does not match any of the proxy's method definitions, so it falls through to the `match` clause (02), which receives messages reflectively. The clause's head is a pattern matched against a pair of the message name (here, `"foo"`) and the list of arguments (here, a list holding only proxy P2).

P1 asks for the value of P2's `stub` and `whoBlame` fields, which hold S2 and Carol's Who (03–05). (To protect against misbehaving app-objects, P1 actually does this by rights amplification[11] rather than the `getGuts` message shown here.) P1 then sends `intro(whoBob)` to S2 (06), by which Alice is saying in effect *"Carol, I'd like to share with Bob my access to C. Could you create a stub for Bob's use?"* Nothing forces Alice to share her rights in this indirect way; Alice's P1 *could* just give Bob direct access to S2. But then Carol would *necessarily* blame Alice for Bob's use of S2, which Alice might not like.

Carol makes S3 for Bob's use of C (08). Carol tags S3 with Bob's Who, so Carol can blame Bob for messages sent to S3. Carol then "gift wraps" S3 for Bob and returns `gs3`, the gift-wrapped S3, to Alice as the result of the `intro` message (09–11). Alice includes `gs3` in the `p3Desc` record encoding the p2 argument of the original message (12). By including this in the `deliver` request to Bob's S1 (13), Alice is saying in effect *"Bob, please unwrap this to get the ability to use an object provided by Carol."*

Bob's S1 unpacks the record (15), unwraps `gs3` to get S3 (16–26), which it uses to make proxy P3 (27). Bob tags P3 with Carol's Who, so Bob can blame Carol for the behavior of S3. S1 then includes P3 as the argument of the app-level `foo` message it sends to B using E's reflective `E.call` primitive (28).

Clearly the `unwrap` function must be the inverse of the `wrap` function. Identity functions would be simple, but would also give Alice's P1 access to S3. Were all Horton objects provided by a mutually trusted platform, Bob and Carol could rely on P1 not to abuse its access to S3, and we'd be done. But this would preclude using Horton between mutually suspicious machines. Given that P1 behaves as Alice wishes, P1's access to S3 would let Alice fool Carol into blaming Bob for messages Alice sends to S3.

Carol's S2 should at least gift-wrap S3 so only Bob can unwrap it. Could we simply use the `seal`/`unseal` operations of Bob's who/be pair as the `wrap`/`unwrap` functions? Unfortunately, this would still enable Alice to give Bob a gift allegedly from Carol, but which Bob unwraps to obtain a faux S3 created by Alice.

In our solution, Carol's `wrap` creates a `provide` function, seals it so only Bob can unseal it, and returns the resulting box as the wrapped gift (11). Bob's `unwrap` unseals it to get a `provide` function allegedly from Carol (18). Bob will need to call

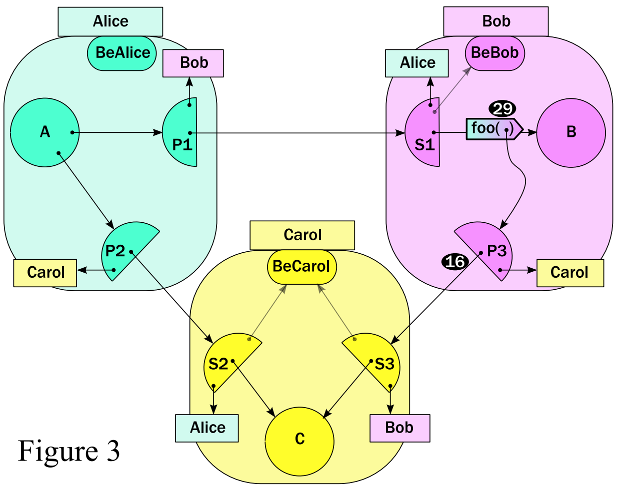

Figure 3

```
01:   # A says:... b.foo(c) ...

def makeProxy(whoBlame, stub) {
  return def proxy {
04: to getGuts() {                  # as P2
05:   return [stub, whoBlame]}
02: match [verb, [p2]] {            # as P1
03:   def [s2, whoCarol] := p2.getGuts()
06:   def gs3 := s2.intro(whoBlame)
12:   def p3Desc := [gs3, whoCarol]
13:   stub.deliver(verb, [p3Desc])}}}

def makeStub(beMe, whoBlame, targ) {
  return def stub {
07: to intro(whoBob) {              # as S2
08:   def s3 := makeStub(beMe,whoBob,targ)
09:   return wrap(s3, whoBob, beMe)}
14: to deliver(verb, [p3Desc]) {    # as S1
15:   def [gs3, whoCarol] := p3Desc
16:   def s3 := unwrap(gs3, whoCarol, beMe)
27:   def p3 := makeProxy(whoCarol, s3)
28:   E.call(targ, verb, [p3])}}}

29: # B implements:... to foo(c) {...} ...

10: def wrap(s3, whoBob, beCarol) { # as S2
22:   def provide(fillBox) {
23:     def fill := beCarol.unseal(fillBox)
24:     fill(s3)}
11:   return whoBob.seal(provide)}
17: def unwrap(gs3,whoCarol,beBob){ # as S1
18:   def provide := beBob.unseal(gs3)
19:   var result := null
25:   def fill(s3) {result := s3}
20:   def fillBox := whoCarol.seal(fill)
21:   provide(fillBox)
26:   return result}
```

`provide` (21) so that *only* Carol can provide S3 to him. Bob declares an assignable `result` variable (19), and a `fill` function for Carol to call to set this variable to S3. He seals this in a box only Carol can unseal (20) and passes this to `provide` (21). Carol's `provide` unseals it to get Bob's `fill` function (23), which Carol can call to set the `result` to S3 (24–25). After Carol's `provide` returns, Bob's `unwrap` returns whatever it finds in the `result` variable (26).

Should Bob and Carol ever come to know each other independently of Alice, they can then blame each other, rather than Alice, for actions logged by P3 and S3. Say C is a wiki page. If Carol believes that Bob is not a pseudonym for Alice, and Carol decides that Bob has abused this page, Carol should then revoke Bob's access without revoking Alice's access by shutting off S3. If Bob decides that C is flaky, he can stop using Carol's services by shutting off proxies such as P3. This completes the induction.

3 Related Work

Some distributed ocap systems interpose objects to serialize/unserialize messages[2, 13], stretching the reference graph between local ocap systems. Secure Network Objects[20] and Client Utility[7] leveraged their intermediation to add some identity tracking. Horton unbundles such identity-based control as a separately composable abstraction.

Reactive security ocap patterns include the logging forwarder[17] and the caretaker[12]. But these patterns provide no means for Alice to ask Carol to issue an access path which Carol can attribute to Bob.

Petmail[22] and SPKI[3] provide some features similar to Horton in a non-ocap environment. They also show how petnames[18] can enable secure human interpretation of the identities. Future Horton extensions should similarly support petnames.

4 Conclusions

Delegation is fundamental to human society. If digital systems are to mediate ever more of our interactions, we must be able to delegate responsibility within them. While some systems support the controlled delegation of authority, and other systems support assignment of responsibility, today we have no means for delegating responsibility, that is, delegating authority coupled with assigning responsibility for using that authority. Horton shows how delegation of responsibility can be added to systems that already support delegation of authority—object-capability systems.

We thank the e-lang and cap-talk communities, especially Tyler Close, Terence Kelly, Charles Landau, Sandro Magi, Kevin Reid, Terry Stanley, Marc Stiegler, Brian Warner, and Meng Weng Wong.

References

[1] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.

[2] J. E. Donnelley. A Distributed Capability Computing System. In *Proc. Third International Conference on Computer Communication*, pages 432–440, Toronto, Canada, 1976.

[3] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory (IETF RFC 2693), Sept. 1999.

[4] T. S. Geisel. *Horton Hears a Who!* Random House Books for Young Readers, 1954.

[5] V. D. Gligor, J. C. Huskamp, S. Welke, C. Linn, and W. Mayfield. Traditional capability-based systems: An analysis of their ability to meet the trusted computer security evaluation criteria. Technical report, National Computer Security Center, Institute for Defense Analysis, 1987.

[6] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, Apr. 1984. IEEE.

[7] A. H. Karp, R. Gupta, G. Rozas, and A. Banerji. The Client Utility Architecture: The Precursor to E-Speak. Technical Report HPL-2001-136, Hewlett Packard Laboratories, June 09 2001.

[8] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[9] 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.

[10] M. S. Miller, K.-P. Yee, and J. S. Shapiro. Capability Myths Demolished. Technical Report Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, mar 2003.

[11] J. H. Morris, Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, 1973.

[12] D. D. Redell. *Naming and Protection in Extensible Operating Systems.* PhD thesis, Department of Computer Science, University of California at Berkeley, Nov. 1974.

[13] R. D. Sansom, D. P. Julin, and R. F. Rashid. Extending a Capability Based System into a Network Environment. In *Proc. 1986 ACM SIGCOMM Conference*, pages 265–274, Aug. 1986.

[14] M. Seaborn. Plash: The Principle of Least Authority Shell, 2005. `plash.beasts.org/`.

[15] J. S. Shapiro and S. Weber. Verifying the EROS Confinement Mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, 2000.

[16] A. Spiessens. *Patterns of Safe Collaboration.* PhD thesis, Université catholique de Louvain, Louvain la Neuve, Belgium, February 2007.

[17] M. Stiegler. A picturebook of secure cooperation, 2004. `erights.org/talks/efun/SecurityPictureBook.pdf`.

[18] M. Stiegler. An Introduction to Petname Systems. In *Advances in Financial Cryptography Volume 2.* Ian Grigg, 2005.

[19] M. Stiegler, A. H. Karp, K.-P. Yee, T. Close, and M. S. Miller. Polaris: Virus-safe Computing for Windows XP. *Commun. ACM*, 49(9):83–88, 2006.

[20] L. van Doorn, M. Abadi, M. Burrows, and E. P. Wobber. Secure Network Objects. In *Proc. 1996 IEEE Symposium on Security and Privacy*, pages 211–221, 1996.

[21] D. Wagner and E. D. Tribble. A Security Analysis of the Combex DarpaBrowser Architecture, Mar. 2002. `combex.com/papers/darpa-review/`.

[22] B. Warner. Petmail. *Codecon*, 2004. `petmail.lothar.com`.