Non-Delegatable Authorities in Capability Systems

Toby Murray Duncan Grove

Command, Control, Communications and Intelligence Division,
Defence Science and Technology Organisation,
PO BOX 1500, Edinburgh, South Australia 5111

toby.murray@comlab.ox.ac.uk, duncan.grove@dsto.defence.gov.au

Abstract

We present a novel technique, known as the *non-delegatable authority* (NDA), for distributing authority to unconfined subjects in capability systems that prevents them from sharing the exact same authority that they have been given with anyone else. This feature is present in common systems based on access control lists (ACLs) in which one may hand out a permission without handing out the associated "grant" right, but has been thought to be impossible to express in capability systems until now. Consequently, we demonstrate that NDAs may be used to express ACL-like constructs and their basic pattern is directly applicable for implementing Multi-Level Security and identity-based access controls in the object-capability model.

The extra complexity introduced by our NDA implementation can be hidden behind constructs that allow NDAs to be wielded as if they were ordinary capabilities to the target resource. These constructs cannot break the non-delegatability constraint and allow NDAs to be used effectively, although with less efficiency than delegatable authorities.

1 Introduction

It has been argued that the inability of capability systems to prevent *delegation* by capability passing hinders security because it increases the burden of trust (as argued in [31]) and makes it difficult to support mandatory access controls [1] (as argued in [13]), including enforcing confinement [10] (as argued in [9]) and the *-property [3] (as argued in [7]). Despite these claims, working implementations of capability systems able to enforce mandatory access controls (such as [22]) and confinement (such as [21, 25]) have existed for decades, and recent work has formally proved that confinement can in fact be guaranteed [26] and that the *-property can indeed be enforced [18, 27]. However, all of these techniques rely on being able to limit, or *confine*, the outward communication channels of untrustworthy subjects. Unfortunately, this assumption is overly restrictive in certain circumstances. For example, none of these techniques can be applied to allow one to distribute authority to an *unconfined* subject already in existence, to whom arbitrary outward communications channels might be available, while preventing that authority from being delegatable. This paper describes a novel technique that addresses this weakness and can be used alongside the existing mechanisms listed above to tightly control the sharing of authority in capability-based systems.

Our technique is known as the *non-delegatable authority* (NDA), and allows some entity A to give an unconfined entity B authority to access some resource C, while ensuring that B cannot pass on that exact same authority to any other entity. We present our implementation within the context of the object-capability model [20], although it can be generalised to many real-world capability systems including E [15], EROS [25], W7 [23] and the Annex Capability System [8]. The object-capability model accurately captures the semantics of many capability systems (such as those mentioned previously as well as Dennis and Van Horn's original supervisor [4]), framing the

principles of capability security [4] within the terms of contemporary object-based systems. This model has also been used as a base for reasoning about authority within capability systems [28].

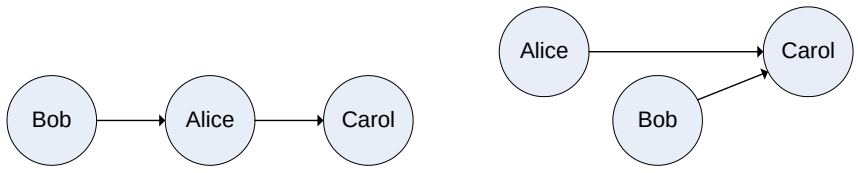
In the object-capability model, capabilities address *objects* and provide the sole means by which objects may interact through *method invocation*. Here, the object graph also serves as the access graph since capability possession is the sole requirement for access. The golden rule of the object-capability model is that "only connectivity begets connectivity" [17]. This rule requires that capabilities may only be passed between objects that are already connected within the object graph, perhaps through other intermediate objects. Capabilities may be passed either as method parameters, or may be returned as the result of an invocation.

The model places no further restrictions on the flow of capabilities. This means that any object that possesses a capability can freely pass it to any other object that it possesses a capability to, or to any other object that possesses a capability to it. As previously stated, we refer to this act as *delegation*, which enables one object to conveniently pass part of its authority to another. Proponents of capability systems have argued against the notion that delegation is harmful for security. Instead, they posit that it is a feature that necessarily enhances security [29], as it enables the natural implementation of systems that preserve the principle of least authority [19], refined from the principle of least privilege [24]. It has also been asserted that preventing delegation is not only harmful for security but also unnecessary [5]. This theory rests on the premise that no conventional access control system can prevent certain types of authority sharing amongst unconfined subjects. The following section explores this idea in order to lay the foundations for an examination of the possible value that non-delegatable authorities might provide.

1.1 Authority Sharing

We explicitly acknowledge that there are hard limits to the extent to which the sharing of authority by unconfined subjects can be controlled in conventional access control systems such as those employing capabilities [4] and access control lists [11]. Specifically, in none of these systems can an unconfined subject be prevented from sharing part of its authority with any other with whom it has a bidirectional communications link[1]. If one particular subject is given access to a resource, it can always share its authority with other subjects by acting as a proxy [6]. While proxying and delegation both constitute sharing of authority, they are not the same.

A proxy is by definition interposed between the subject and the resource, meaning that they can at any time revoke the subject's authority by refusing to proxy and generally interfere with the subject's access to the resource. Figure 1(a) depicts Alice sharing her authority with Bob to access the target resource Carol by proxying. When proxying, Alice cannot ever give Bob the exact same level of authority that she possesses over Carol, so Alice always retains more authority than Bob. On the other hand, if Alice delegates her authority over Carol to Bob by passing him her capability, she gives Bob direct access to Carol. Unlike when proxying, Alice cannot interfere with this access and Bob is given exactly the same authority over Carol that Alice has. Figure 1 depicts the difference between proxying and delegating.



(a) When proxying, Alice retains more authority than Bob

(b) When delegating, Alice and Bob share identical authority

Figure 1: The difference between proxying and delegating

[1]Note that in the object-capability model, a bidirectional communications link is not necessary. An outward communications link (i.e. a capability to another object) can be used to bootstrap a bidirectional channel.

An NDA provides a means of distributing authority to unconfined subjects while ensuring that the only way it can be shared is analogous to proxying. Its purpose is to prevent an unconfined subject from being able to share exactly the same authority that they have been given with anyone else.

1.2 The Value of Non-Delegatable Authority

One possible use for non-delegatable authorities is to guard against unreliable software that might be prone to exploitation by an attacker. Consider a scenario where Alice is granted non-delegatable authority to access Carol. In order to share this authority with Bob, Alice must act in a manner analogous to proxying, actively collaborating with Bob at each invocation. If Alice is collaborating only because Bob has confused her into doing so (perhaps because Bob has exploited a vulnerability in Alice's code) then Alice must remain confused in order for Bob to retain access. On the other hand, if Alice is able to share her authority with Bob by delegating, she need only be confused into delegating once. In this case Bob has effectively stolen her capability and will retain the authority to access Carol regardless of whether Alice remains confused. In particular, if the vulnerability in Alice's code is subsequently patched, Bob will still retain access. If Alice is only able to share by proxying, however, then patching the vulnerability also revokes Bob's authority to access Carol. As such, the value of non-delegatability increases with the likelihood of confusion and unreliability. We return to this theme in Section 3, in which we show by example how non-delegatable authorities can render capability-stealing attacks useless.

Non-delegatable authorities can also be used to prevent unintended *rights amplification* [17]. Rights amplification occurs when a subject uses two capabilities together in order to obtain more authority than the sum of the authority provided by each individually. Consider the scenario where Alice and Bob each hold one half of a pair of capabilities, which when used together cause rights amplification. Alice cannot achieve rights amplification unless she holds both capabilities simultaneously. Preventing Bob from delegating his capability to Alice makes it impossible for Alice to achieve rights amplification. The same applies in reverse to Bob. Hence, rights amplification is impossible when both Alice and Bob are prevented from delegating to one another.

Informally, preventing an unconfined subject from delegating and thus forcing it to proxy also makes it unreasonable for that subject to deny responsibility for, or knowledge of, how their authority is used by others with whom they choose to share it. The very impossibility of preventing sharing altogether makes it imperative that subjects can be held to account for how their authority is used on their behalf. Other mechanisms for maintaining accountability in the face of sharing include the *Logger* pattern described in [29]. In contrast to the Logger pattern, when forced to proxy because they cannot delegate, subjects have the chance to actively intercede to ensure that their authority is being used appropriately. Thus, the value of non-delegatability might be two-fold: when giving Alice a non-delegatable authority, we not only reduce the degree to which we must trust her, but also the degree to which Alice must trust anyone else with whom she chooses to share it.

The idea that non-delegatable authority increases accountability is not surprising given that it pervades many real-world applications for which audit and accountability are primary concerns. Examples include financial policies that require spending to be approved by a particular officer, *originator controlled release* [14] policies that require all access to data to be approved by its originator, the Lockean *non-delegation doctrine* arguing that "the legislative neither must nor can transfer the power of making laws to anybody else, or place it anywhere but where the people have" [12] (quoted in [30]), as well as numerous other policies that specifically limit the authority to delegate. For these reasons, being able to express non-delegatable authorities in the object-capability model naturally increases its scope and applicability.

1.3 Sources of Authority

If capability possession were the only source of authority that the object-capability model provided, then it would be impossible to construct non-delegatable authorities. The reason for this is simple:

if all authority flowed from the capabilities one possessed, then all authority could be delegated since all capabilities can be delegated. If one believes that method invocation is the only way to exercise authority, then it appears natural to conclude that capabilities must be the sole source of authority. However, our contribution is to recognise that there is another way to exercise authority within the object-capability model, leading to another source of authority. In certain object-capability implementations this authority cannot be delegated. In these systems we may harness this source of authority to build general purpose non-delegatable authorities.

1.4 Authority by Method Return

This new source of authority is made visible by recognising the influence an object may have over its invoker. Depending on the behaviour of the invoker, which is defined by its code, this influence can be significant. Some objects can wield authority over their invoker by the simple act of what they return in response to method calls. It is straightforward to reason that if when one object A invokes another B causing B to invoke C, that A has wielded authority to invoke C. By the same logic, we argue that if B's response to an invocation by A causes A to invoke C, that B has wielded authority to invoke C. When an invoked object influences the behaviour of its invoker, causing the invoker to exercise some of its own authority, this authority has been wielded by the invoked object.

Importantly, in many object-capability systems the general authority to reply is not delegatable. Examples include systems such as W7 [23] and the Annex Capability System [8] that never reify this authority, making it impossible to delegate. Other examples include systems such as E [15] and EROS [25] that reify the authority to reply only to the *current* invocation in the form of a "use once" capability. In these systems, an object has no means to delegate the authority to reply to *all* such invocations. The only way it may share this authority is by re-delegating the "use once" reply capability each time it is invoked. Under these circumstances, an object is forced to actively collaborate at each invocation in order to share its authority, thereby acting in a manner analogous to a proxy. Hence, even in these systems, the general authority to reply is not delegatable.

2 Implementing Non-Delegatable Authorities

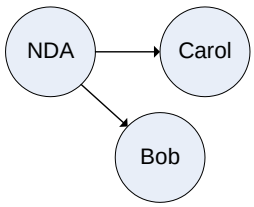


Figure 2: NDA gives Bob non-delegatable authority to invoke Carol

Figure 2 depicts an object, NDA, that has access to both Bob and Carol. Alice (not depicted) wishes to give Bob non-delegatable authority to invoke methods on Carol. She has constructed NDA and given Bob a capability to it that he can use to access Carol. When invoked, NDA invokes Bob whose response explicitly influences NDA and may cause NDA to invoke Carol. Bob's response also influences the method that NDA invokes on Carol and the parameters that are passed. In order to allow Bob to wield authority effectively, NDA expects him to return 2 items from the invocation. The first denotes the method that NDA is to invoke on Carol, the second is a list of parameters that NDA will pass to Carol. NDA then invokes the given method on Carol, passing it the parameters that Bob has specified. In order to allow Bob to receive the

results of his invocation, NDA can invoke Bob after it has invoked Carol, passing the results back to Bob as parameters to the invocation.

Usually, it is expected that Bob will invoke NDA in the first instance when he wishes to invoke Carol, although this need not be the case. NDA's purpose is to ensure that no matter who invokes NDA, Bob will still have control over the invocation of Carol and the first look at the returned results. Therefore, no matter how far Bob passes his NDA capability, he can never give anyone else the same authority to access Carol that he has. NDA prevents Bob from being able to delegate his authority to access Carol even if he delegates his authority to access NDA, because NDA always keeps Bob interposed. Put another way, NDA ensures that Bob can only share his authority over Carol in a manner that is analogous to proxying.

A possible implementation for NDA is shown in Figure 3. All code examples throughout this paper appear in the object-capability language *E* [15] and assume that objects are co-located. E was chosen because it is the most mature and unambiguous formalism available for describing object-capability design patterns. Readers unfamiliar with E may refer to Appendix A, which contains an overview of the subset of E used in this paper.

```
def makeNDA(subject, target) :any {

    def nda() :any {

        # nda passes a reference to itself when it invokes the subject
        def [methodName, args] := subject.getNDAInvoc(nda)

        # null method name means 'subject' is unwilling to invoke 'target'
        if (methodName != null){

            # call 'methodName' on 'target', passing 'args'
            def response := E.call(target, methodName, args)

            # nda passes itself again, when invoking the subject
            subject.recvNDAResponse(nda, response)
        }
    }

    return nda
}
```

Figure 3: An implementation of NDA

Using this implementation, Alice can construct a non-delegatable authority for Bob to access Carol by calling `nda := makeNDA(bob, carol)`. When invoked, NDA first invokes the `getNDAInvoc()` method on Bob to get the method name and parameters to invoke on Carol. If the method name is non-null, then the method is invoked on Carol before returning the results to Bob by calling the `recvNDAResponse()` method. NDA passes a reference to itself when invoking Bob to allow Bob to infer the authority he is wielding when he returns a non-null method name, and to allow Bob to tie the response back to the original invocation.

Our NDA implementation necessarily increases the level of complexity. As such, we need constructs that hide the extra complexity if it is going to be practical. The next section provides examples of appropriate constructs that can be used to achieve this purpose.

2.1 Wielding Non-Delegatable Authorities

Continuing with the previous example, for Bob to make effective use of NDA, his `getNDAInvoc()` method must be implemented so that it only responds with a non-null method name if he actually

wishes to invoke a method on Carol at the present time. Bob should also be able to wield many different NDAs without becoming confused. Bob may achieve this by maintaining an *invocation map*. This is a mapping from NDA objects to invocation requests, where an invocation request is a pair: (method name, method parameters). When wishing to invoke a method using a particular NDA *nda*, Bob adds a mapping $nda \rightarrow (methodName, params)$ to the invocation map and then invokes *nda*. When Bob's `getNDAInvoc()` method is invoked, it simply checks the invocation map for whether there is a mapping for the NDA object passed in. If so, it returns the invocation but otherwise returns a null method name.

Bob may also maintain a *response map* from NDA objects to invocation responses, where an invocation response is simply a capability to the values returned from an invocation. When invoked, Bob's `recvNDAResponse()` method checks whether there is an invocation mapping for the NDA object *nda* passed in and if so, adds a mapping $nda \rightarrow response$ to the response map before removing the corresponding mapping from the invocation map. Once Bob's original invocation of NDA returns, Bob simply checks the response map in order to access the response returned from the invocation.

Figure 4 depicts a possible partial implementation for Bob, including his `getNDAInvoc()` and `recvNDAResponse()` methods, as just described.

Using this implementation, Bob can then wield NDA, denoted `nda`, to call a method `methodName` on Carol with the list of arguments `args` and bind the name `results` to the response, as follows:

```
# add invocation mapping for 'nda'
ndaInvocs[nda] := makeNDAInvoc(methodName, args)

# invoke 'nda'
nda()

# retrieve response from response map
results := ndaReturns[nda]
```

Trying to add more than one invocation mapping for a single NDA invocation will overwrite the original mapping. Also, trying to invoke NDA without adding a new invocation mapping will cause `getNDAInvoc()` to return null method names, preventing any invocation on the target object. For these reasons, the above three steps should always be performed together.

Of course, Bob can create an object to handle the three steps, ensuring that they are always performed together while providing a convenient interface for wielding an NDA. A possible implementation for this object, which we call an NDAWrapper, appears in Figure 5.

Bob can wield an NDAWrapper as if it were a regular capability to the target object. For example, suppose Alice has given Bob an NDA, denoted `nda`, for him to access Carol. Bob can invoke Carol's `incr()` and `decr()` methods, which increment and decrement their argument respectively, as follows:

```
def ndaWrapper := makeNDAWrapper(nda)
if (ndaWrapper.incr(4) != 5){
   # handle error ...
}
if (ndaWrapper.decr(9) != 8){
   # handler error ...
}
```

At all times Bob remains interposed, preventing him from delegating his authority to invoke Carol, despite the fact that Bob can invoke NDAwrapper as if it were a direct reference to Carol and delegate his capability to NDAWrapper. Passing NDAWrapper is analogous to sharing authority by proxying and Bob can never give any other subject the same authority that he has over Carol.

Bob has more authority than any other object to whom he may pass NDAWrapper simply because of who he is. Even if Bob gives every capability that he possesses to some other object

```
def makeBob() :any {
    var ndaInvocs := [].asMap().diverge()  # invocation map
    var ndaReturns := [].asMap().diverge() # response map

    def makeNDAInvoc(methodName, args) :any {
        def ndaInvoc {
            to getMethodName() :any { return methodName }
            to getArgs() :any { return args }
        }
        return ndaInvoc
    }

    def bob {
        # ... code for Bob's other methods ...

        to getNDAInvoc(nda) :any {
            # see if this 'nda' maps to any invocation
            if (ndaInvocs.maps(nda)){
                def invoc := ndaInvocs[nda]
                return [invoc.getMethodName(), invoc.getArgs()]
            }
            return [null, null]
        }

        to recvNDAResponse(nda, response) :void {
            # if no invocation exists for this 'nda', ignore the response
            if (ndaInvocs.maps(nda)){
                ndaReturns[nda] := response  # add response mapping
                ndaInvocs.removeKey(nda)     # remove invocation mapping
            }
        }
    }
    return bob
}
```

Figure 4: Partial implementation of Bob

Dennis, he still wields more authority over Carol than what Dennis ever could. While this is precisely the purpose of an NDA as we explained earlier in section 1.1, it also highlights the fact that Bob now wields authority that comes to him on the basis of his identity, as well as the capabilities he possesses. We explore this issue further in Section 3 in which we discuss the implementation of two forms of identity-based access control.

2.2 Costs of Non-Delegatable Authorities

Each invocation on NDAWrapper results in two invocations on the subject, one invocation on NDA and one invocation on the target. Hence, the NDAWrapper adds four times the ordinary overhead of a direct delegatable reference to the target. This overhead might be acceptable for object references that do not cross machine boundaries but is likely to add significant latency if the objects are widely distributed. Another cost for distributed environments is decreased reliability, since the subject (Bob) cannot access the target (Carol) unless the NDA is reachable[2]. The choice

[2]We presume that the NDAWrapper is co-located with the subject and is therefore reachable.

```
def makeBob {
    var ndaInvocs := [].asMap().diverge()  # invocation map
    var ndaReturns := [].asMap().diverge() # response map

    # ... other definitions as above ...

    def makeNDAWrapper(nda) :any {
        def ndaWrapper {
            # match any method called
            match [methodName, args] {
               ndaInvocs[nda] := makeNDAInvoc(methodName, args)
               nda()
               ndaReturns[nda]  # implicit return
            }
        }
        return ndaWrapper
    }

    # ... definition for bob as above ...
}
```

Figure 5: NDAWrapper implementation

of whether to place the NDA at the site of the subject or the target has obvious implications for performance and reliability. Unfortunately, these are the costs that must be paid for the advantages of being able to express and enforce non-delegatability while hiding its inherent complexity.

2.3 Sharing Non-Delegatable Authorities

Bob has two choices when wishing to usefully share his non-delegatable authority to Carol. Passing NDA is unlikely to be useful unless Bob also passes capabilities to his invocation and response maps. Instead, Bob can achieve the same result by passing a capability to NDAWrapper, which is not only simpler but more reliable. On the other hand, Bob may choose to create a new NDA to NDAWrapper, in which case he is passing non-delegatable authority to his non-delegatable authority to Carol. We refer to this as NDA *chaining*.

NDA chaining might be particularly useful for distributing authority throughout a natural hierarchy, since the NDA chain implicitly models the chain of command. For example, suppose Bob is Alice's subordinate and that Bob is responsible for managing Dennis. The chain of command is Dennis, Bob, Alice. Bob chooses to create a new NDA `nda2` for Dennis to his NDAWrapper `ndaWrapper` and passes Dennis `nda2`. Dennis creates a new NDAWrapper `ndaWrapper2` to enable him to wield `nda2` effectively. The chain of invocations that will occur when `ndaWrapper2` is invoked are: `ndaWrapper2` invokes `nda2`, `nda2` invokes Dennis, `nda2` invokes `ndaWrapper`, `ndaWrapper` invokes `nda`, `nda` invokes Bob, `nda` invokes Carol. The advantage is that if Dennis chooses to share his authority to Carol using either technique, that he is still held accountable when it is exercised because he remains within the chain of invocation. Thus, NDA chaining facilitates auditing. Unfortunately, chaining NDAs necessarily decreases performance and potential reliability.

3 Identity-Based Access Controls

The NDA works because it binds the authority to invoke methods on the target object (Carol) directly to its holder (Bob). This binding is provided by two properties of the object-capability

system: (1) that when NDA invokes its capability to Bob, that Bob is actually the object that gets invoked and not some other, and (2) that Bob cannot pass on the general right to reply to all such invocation to any other object, thereby forcing him to actively collaborate each time NDA invokes him in order to share his authority to invoke Carol. Binding an authority to some particular object enables the implementation of identity-based access controls, such as access control lists [11] and credentials.

We now describe how both might be implemented using NDAs and the underlying principle of authority by method return. In doing so we do not wish to suggest that either example is an improvement over existing credential or ACL systems, but only that it is indeed possible to express these concepts within the object-capability model. To our knowledge, this has never been demonstrated before.

3.1 Expressing ACL-like Controls

In our example, the NDA gives Bob more authority than he can share simply because of who he is. It can be thought of as an identity-based access control that allows only Bob general access to invoke Carol. In this sense, it is similar to an ACL that specifies that (only) Bob is allowed to invoke any method on Carol[3]. As with an ACL, Bob may share the authority to invoke Carol by acting in a manner analogous to a proxy but cannot delegate it to anyone else.

Suppose NDA was extended, however, to replace the single reference to Bob with a list of references to subjects. When invoked, NDA would take a single parameter denoting the subject that wishes to wield authority to invoke Carol. At this point, NDA would check that this subject was in the list and if so, invoke it in the same way that it invoked Bob before. This would provide the implementation for an ACL-like control that specifies that all of the subjects in the list can invoke any method on Carol.

NDA can be extended further so that for each subject in the list, NDA also maintains a list of methods that that subject is allowed to call. NDA now behaves as before, except that when the subject returns the result of the `getNDAInvoc()` method, NDA checks that the method name is on the list of methods that this subject is allowed to call and if not, does not invoke Carol. NDA now behaves as an ACL-like control that specifies a set of subjects that can access Carol, and for each subject specifies the methods they are allowed to invoke. As such, NDAs can be used to implement fine-grained ACL-like access controls for individual objects.

3.2 Expressing Credentials

As a final demonstration of the practicality of the NDA pattern, we now consider a real-world example applicable to the Annex [8] pervasive computing testbed. The Annex system comprises a number of devices, each of which hosts their own object-capability implementation in which password capabilities [2] provide a universal capability representation. Each Annex user is assigned their own device, which is irreversibly bound to their identity.

Annex requires users to be able to prove their security clearance before being able to access sensitive or classified services. For example, when Bob attaches his device to a SECRET network router R, he must first prove to R that he is SECRET-cleared before it will forward his traffic. Bob's clearance is very much like a non-delegatable authority, except that it does not refer to any specific object that Bob is to access. Instead, a security clearance is a general statement about Bob's trustworthiness. Like an NDA, it is inextricably bound to Bob, to whom it gives authority. The similarity between the user's clearance and an NDA allows us to implement clearance-embodying objects, called *credentials*, using the same idea employed by the NDA.

Credentials are issued and hosted by a security authority (SA). Every device has a SA capability that they can use to verify the authenticity of credentials. Bob can prove to R that he is SECRET-cleared by passing it a capability to his credential object. R can then query SA to verify the authenticity of Bob's credential and to learn whether Bob is SECRET-cleared. To prevent some

[3]In this imaginary ACL, we presume that Bob does not also possess the "grant" right and is hence unable to share his authority to invoke Carol, except by proxying.

device that is not SECRET-cleared learning that Bob is SECRET-cleared, SA will only divulge whether one credential object carries at least the same clearances as another. For R to learn that Bob is SECRET cleared it has to pass its own credential capability to SA along with Bob's, at which point it can infer whether Bob is cleared at least as high as R. Credentials support methods such as `getUserName()` and `getDeviceID()`. This final method returns the unique identifier assigned to the user's device at its time of manufacture. This identifier is the only piece of information required to route network traffic to the device, and allows one to infer its (current) network address [8].

Just like the NDA object in our example, Bob's credential object also has a capability that points to Bob. This capability addresses an object on Bob's device, known generally as the UserProxy, since it is responsible for acting as a proxy for Bob. The credential object supports a method `getUserProxy()` to retrieve this capability.

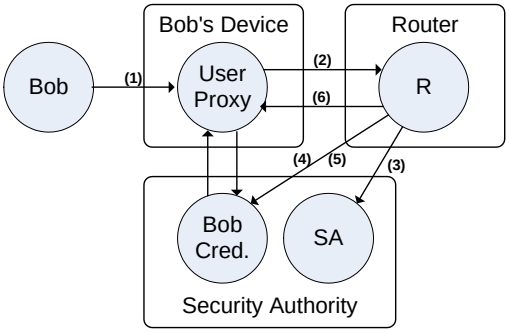


Figure 6: Gaining access to a classified network

The process for Bob gaining access to R is depicted in Figure 6 where arrows represent capabilities and numbers denote invocations on the corresponding capability. Bob begins by (1) invoking the `accessNetwork()` method of his UserProxy object, passing it a capability to R. This capability could have been given to Bob over a number of different distribution channels, including being sent on router advertisements. Bob's UserProxy responds by (2) invoking R, passing it a capability to Bob's credential object. R then (3) invokes SA to check the validity of the credential and whether Bob has adequate clearance before (4) invoking its `getDeviceID()` and (5) `getUserProxy()` methods. R now has enough information to determine Bob's current network address as well as a capability to Bob's UserProxy object. R then (6) invokes Bob's UserProxy object, asking it if Bob wishes to access the network. R should pass a capability to itself with this invocation, which Bob's UserProxy should compare against the original capability that it invoked. This invocation is analogous to the `getNDAInvoc()` invocation that NDA makes on Bob in our previous example. If Bob's UserProxy object responds affirmatively, then R gives Bob access to the network by adding a firewall rule allowing traffic to and from Bob's network address.

The use of the NDA pattern in this final example prevents an attacker who steals all of Bob's capabilities from being able to access the SECRET network on Bob's credentials. We noted in the previous section that the NDA prevents Bob from being able to pass on his exact authority, even if he gives away all of his capabilities. Employing the same pattern here to provide Bob with authority not embodied in his capabilities naturally prevents this attack from succeeding. The NDA pattern binds the authority implied by Bob's SECRET clearance to his identity, so it can only be stolen by impersonating Bob. Given that no conventional identity-based access control systems can function in the face of successful impersonation and the infeasibility of impersonation, we consider this to be outside of our threat model.

4 Conclusion

We have presented a means for constructing non-delegatable authorities that can be applied in certain object-capability systems. Our technique, known as the NDA, is built upon the general notion of authority by method return and relies on this not being delegatable. NDAs enable authority to be distributed, even to unconfined subjects, in such a way that the recipient is unable to share the exact same authority that they have been given with any other subject. This extends the expressiveness of the object-capability model by allowing one to describe and enforce security policies that were previously thought impossible, even by capability proponents [16]. These include those based on the notion of non-delegation and those that grant authority based on a subject's identity.

NDAs provide an alternative between having to distribute authority to untrusted subjects without any means to prevent it from being delegated and having to rule out delegation in general. Both of these choices can be counterproductive for security. Instead, our NDA mechanism allows one to make an appropriate choice about how to distribute authority without having to sacrifice security.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and insights. We would also like to thank Chris North, Chris Owen, Mark Miller and Kevin Reid who provided valuable feedback on early drafts of this paper. Any errors or omissions that remain are our own.

References

[1] James P. Anderson. Computer security technology planning study, Volume 2. Technical Report ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford MA 01730, October 1972.

[2] M. Anderson, R. D. Pose, and C. S. Wallace. A password capability system. *The Computer Journal*, 29(1):1–8, 1986.

[3] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and MULTICS interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford MA 01730, March 1976.

[4] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–154, March 1966.

[5] James E. Donnelley. Managing domains in a network operating system. In *Proceedings of the Local Networks and Distributed Office Systems Conference*, May 1981.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[8] Duncan Grove, Toby Murray, Chris Owen, Chris North, Jeremy Jones, M. R. Beaumont, and B. D. Hopkins. An overview of the Annex system. To appear in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07)*, 2007.

[9] Paul A. Karger. Improving security and performance for capability systems. Technical Report 149, University of Cambridge Computer Laboratory, 1988. (PhD dissertation).

[10] Butler Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[11] Butler Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[12] John Locke. *The Second Treatise on Government*. 1691.

[13] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information System Security Conference*, pages 303–314, 1998.

[14] Catherine J. McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC: Defining new forms of access control. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 190–200, May 1990.

[15] Mark Miller, Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. E: Open source distributed capabilities, 2001. see `http://www.erights.org`.

[16] Mark S. Miller. Communicating conspirators. see `http://www.erights.org/elib/capability/conspire.html`.

[17] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proceedings of Financial Cryptography 2000*, February 2000. see `http://www.erights.org/elib/capability/ode/index.html`.

[18] Mark S. Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.

[19] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The structure of authority: Why security is not a separable concern. In *Multiparadigm Programming in Mozart/Oz: Proceedings of MOZ 2004, LNCS 3389*, 2005.

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

[21] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. Security in KeyKOS. In *IEEE Symposium on Security and Privacy, 1986*, pages 78–85, 1986.

[22] Susan A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., March 1989. see `http://www.cis.upenn.edu/~KeyKOS`.

[23] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical Report AI Memo No. 1564, MIT, 1996.

[24] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1208–1308, September 1975.

[25] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.

[26] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *Proceedings of the IEEE Symposium on Security and Privacy, 2000*, pages 166–176, 2000.

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

[28] Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in capability-based systems. In *TGC 2005, LNCS 3705*, pages 248–278, 2005.

[29] Marc Stiegler. A picturebook of secure cooperation, 2004. see `http://www.skyhunter.com/marcs/SecurityPictureBook.ppt`.

[30] Nick Szabo. The origins of the non-delegation doctrine. see `http://szabo.best.vwh.net/delegation.pdf`.

[31] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pages 116–128, 1997.

A An Overview of E

In this appendix we include an overview of the subset of E used in this paper. E is a dynamically typed, lexically scoped object language. The `def` keyword is used to define an immutable variable and the `var` keyword is used to define a mutable variable:

```
def five := 5
var num := 5
num := num + 1
```

E has no concept of a class like other object-oriented languages such as Java. Instead, the entire definition of an object is given at instantiation. The following code defines an object `point`, with two methods, `getX()` and `getY()`, each of which can return an object of any type. The values `x` and `y` (below) must be in the lexical scope for the code to be correct.

```
def point {
   to getX() :any { return x }
   to getY() :any { return y }
}
```

An object with a single method `run` is called a *function*. The following shorthand defines the object `makePoint` with a single method `run(x, y)`, which returns a new `point` object.

```
def makePoint(x, y) :any {
   def point {
      to getX() :any { return x }
      to getY() :any { return y }
   }
   return point
}
```

The shorthand `makePoint(3,4)` is equivalent to writing `makePoint.run(3,4)`. We can now create a new point, `myPoint`, with coordinates (3,4) as follows:

```
def myPoint := makePoint(3,4)
```

When defining an object, the `match` keyword can be used to define a method that matches an arbitrary invocation. The expression `E.call` can be used to send an arbitrary invocation to an object. For example, the following code defines an object, `proxy`, which forwards its invocations to the object `target`.

```
def proxy {
   match [methodName, args] {
      E.call(target,methodName,args)
   }
}
```

E has primitive support for lists and maps. The expression `[]` creates a new empty immutable list. The `diverge()` method can be applied to an immutable list (or map) to return a mutable list (or map) whose initial contents match the contents of the immutable list (or map). For example, the following code defines a mutable list `myList` of two elements, `3` and `4`.

```
def myList := [3, 4].diverge()
```

An empty mutable map can be created using the expression `[].asMap().diverge()`. New mappings are added using the following convenient syntax.

```
def myMap := [].asMap().diverge()
myMap[0] := 1   # have 0 map to 1
myMap[1] := 2   # have 1 map to 2
```

Mappings may be queried with similar convenience. The expression `myMap[0]` would evaluate to `1` above. Similarly, `myMap[1]` would evaluate to `2`.