Capabilities on Venti

Adrian Sampson

May 2, 2008

Abstract

This paper concerns the feasibility of implementing a capability security system on the Venti archival storage system. Venti's use of unforgeable, cryptographically secure *fingerprints* as block addresses immediately recalls capability-based security systems. Its write-once property, however, complicates the implementation of a usable read-write capability system on Venti. Three alternative methods for overcoming this limitation are presented and evaluated subjectively. Two solutions require active enforcement by the operating system. One solution, by modifying Venti minimally, succeeds without requiring OS bookkeeping.

1 Background

The Venti archival storage system [2] addresses blocks in a unique way. Rather than using a predictable, serial address for each block in the physical storage, blocks are assigned *fingerprints* when they are written. Fingerprints are cryptographically secure hashes of block contents.

Thus, addresses in Venti are based on block *contents* rather than any external notion of block *location*. Among the consequences of this scheme are a write-once policy (if a block changes, its address changes) and that addresses are unforgeable (as a consequence of being cryptographically secure). The latter property recalls capability-based security systems' requirement for unforgeability. However, the former property of immutability complicates Venti's use as a complete read-write capability system.

A robust capability system for Venti increases the system's viability in contexts beyond enterprise archival storage. Many other applications requiring similar semantics may benefit from a meaningful security system.

1.1 Formulation of Capabilities

Capability-based security systems divide entities into two groups: *subjects* and *objects* [1]. An object is a passive resource that active subjects may access and manipulate. This distinction is intentionally abstract. For the scope of this paper, however, an object will always be a simple repository of data—a linear sequence of bits. A subject will be any active entity whose memory is distinct from other subjects—a process, for instance, or perhaps a user.

The third primitive in a capability-based system is the *capability* itself. A capability is an unforgeable entity that grants subjects access to objects. A capability determines which right it grants and on which object it grants that right. It does not internally determine to which subject it grants that right. Instead, a subject's possession of a capability grants it that capability's rights. For the scope of this paper, only two rights will be considered: reading and writing.

In order to be useful, a capability system must allow transfer of capabilities among subjects. Again for simplicity, we will assume that any subject with capability C may grant C to any other subject. This paper will not consider revocation of capabilities.

A capability system, then, has the following properties:

1. A subject possessing a "read" capability for object O, denoted $C_{O,r}$, may read from O.
2. A subject possessing a "write" capability for object O, denoted $C_{O,w}$, may write to object O.
3. A subject may grant a capability it possesses to another subject.
4. Capabilities are unforgeable. That is, in order for subject S to possess a capability C, another subject must grant C to S or S must be the creator of the object on which C grants rights.

5. A subject may only read or write an object if it possesses the appropriate capability.

Our implementations of capability systems in section 2 will satisfy each of these requirements.

1.2 Objects in Venti

In our proposed system, subjects are independent constructs defined by the operating system such as processes or users. The details and implementation of capabilities will be discussed in section 2. Here we define the final construct, the object, in the context of the Venti archival storage system.

An object must be abstractly interpretable as a sequence of bytes. As proposed in [2], arbitrarily-sized byte sequences may be stored in Venti as trees whose vertices are Venti blocks. A non-leaf node in such a tree is an indirect block containing fingerprints of its children. Leaf nodes are direct blocks and contain the object's data. A tree thus defined will be considered an object in our system. The fingerprint of the tree's root node will be called the object's fingerprint or address.

If the object's data is changed, at least one leaf node in its tree will be changed and its fingerprint updated. The leaf node's parent will, in turn, need to be updated to reflect this change. Inductively, all vertices on the path from the modified leaf node to the root node must be updated. The root node will then have a new fingerprint. Thus, "writing" to an object changes that object's fingerprint.

1.3 Fingerprints as Capabilities

As Venti's authors observe, "a fingerprint does act as a capability since the space of fingerprints is large and the Venti protocol does not include a means of enumerating the blocks on the server" [2]. A naive first approach, then, might be to use the fingerprint itself as a capability.

The requirements for a capability system are listed in section 1.1. Clearly, a fingerprint allows reading the object to which it refers (requirement 1). If subjects may communicate, they may transmit fingerprints to each other, accomplishing the granting of capabilities (requirement 3). Fingerprints, being cryptographically secure hashes, are unforgeable (requirement 4). Relatedly, a fingerprint is required in order to read an object; objects are inaccessible otherwise (requirement 5). The only requirement that is not obviously satisfied is the existence of a "write" capability (requirement 2).

If a subject writes to an object, the newly updated object has a new fingerprint. Other subjects, because they possess the fingerprint of the older version of the object, will not see any changes to the object. Subjects are thus unable to write to objects in a meaningful way: changing an object creates a new object to reflect the changes rather than modifying it in place.

This limitation prevents unadorned Venti fingerprints from usefulness as full capabilities. The remainder of this paper will explore additions to the Venti system to allow its use as a complete capability system.

2 Robust Capabilities

Due to the inherent write-once property of Venti as originally proposed, a useful capability system will require some modification to Venti. Such a system must entail a method for writing to objects that may be shared among subjects. We aim to accomplish this with minimal modifications to Venti's semantics or implementation.

We present, ordered from most naive to most desirable, three alternatives for accomplishing this goal.

2.1 OS-Managed Object Wrappers

The principal problem with writing to objects in Venti is that the object's "address"—its fingerprint—changes whenever its data changes. One obvious approach to this problem is to "wrap" fingerprints with constructs that can be consistently addressed. A wrapper contains the fingerprint of an object but, unlike a Venti block, is accessed in a manner that does not change if the contained fingerprint changes.

Our first approach, in order to avoid modifying Venti itself, places object wrappers in the hands of the operating system. An object wrapper in this case is a data structure stored in a consistently-addressable persistent data store (i.e., a traditional filesystem). To read an object, the OS looks up the fingerprint stored in the wrapper and reads the associated object from Venti. To write an object, it again looks up the fingerprint, performs the modification as specified in section 1.2, and stores the new fingerprint in the wrapper.

Under this system, however, the OS must control access to object wrappers. Without active ac-

cess control, OS-managed object wrappers have no provision for distinguishing subjects that are allowed to read a given object from those allowed to write it. A security system implemented this way, then, does not take advantage of the capability-like properties of Venti itself.

2.2 Block Version Linking

To avoid using a secondary data store aside from Venti, our second approach embeds information into Venti's storage system to support writes to objects. In particular, we extend the "header" information already present for every Venti block.

The data in the header of a Venti block does not affect the block's fingerprint. Thus, we can modify fields headers without changing any addresses in the system. We introduce a new header field, `next-version`. The field's value is initially null for any newly written block. Whenever the system "writes" to this block, the updated content is stored with a new fingerprint as previously discussed. The older block's `next-version` field is then updated to contain the fingerprint of the updated block.

When reading a block, the system first examines the block's `next-version` field. If the field's value is null, the data in the requested block is returned. Otherwise, the block identified by the fingerprint in the `next-version` field is read recursively. By following this fingerprint chain, reading a block will always return the block's most recent version.

A basic implementation of this scheme may require time linear in the number of versions to read a block's most recent version. The system, however, may be optimized in two simple ways. First, when the chain of blocks is traversed, any blocks that do not refer to the most recent version may be updated to do so. This way, the next read of older blocks will require less chain traversal. Second, subjects may be informed of the fingerprint of the most recent block version whenever they attempt to read an older block.

This solution, however, presents the same main drawback as the first design. The operating system is again charged with distinguishing between subjects authorized to read and write a given object. It must implement an access control system in order to prevent subjects from modifying the `next-version` field of blocks to which they do not have the "write" capability.

2.3 Venti-Managed Object Wrappers

To avoid the limitations of the previous two solutions, our final solution must distinguish the "read" and "write" capabilities for any object. We again use "wrapper" constructs that refer indirectly to objects in Venti but this time store them within Venti itself.

As originally proposed, Venti has only one type of block. We propose adding two additional types: read-wrappers and write-wrappers. These blocks differ from ordinary data blocks one important way: their fingerprints are random numbers in the range of the hash function used to generate fingerprints for data blocks. This is necessary because the wrapper blocks have no immutable data that may be hashed to generate a fingerprint. This policy introduces no greater danger of hash collision than an unmodified Venti system.

A write-wrapper stores the fingerprint of an object. A read-wrapper stores the fingerprint of a write-wrapper. The system forbids the reading of wrapper blocks directly; thus, a subject possessing a read-wrapper cannot gain the corresponding write-wrapper.

Reading and writing are supported using the following policies:

- *Reading.* The subject provides the system with the fingerprint of a read-wrapper. The system looks up the fingerprint stored in the read-wrapper. This fingerprint is that of a write-wrapper; the system then looks up the fingerprint stored in this write-wrapper. It reads the data stored at this final fingerprint and returns the result to the requesting subject.

- *Writing.* The subject provides the system with the fingerprint of a write-wrapper and the new data to store. The system stores the data in a new block and writes the block's fingerprint to the provided write-wrapper.

Thus, knowledge of a read-wrapper's fingerprint only allows reading a object; knowledge of a write-wrapper's fingerprint only allows writing to a object. Furthermore, when one subject writes to an object using a write-wrapper, all other subjects possessing the corresponding read-wrapper's fingerprint can observe the changes. These respective fingerprints can therefore be used as capabilities. Because the fingerprints are small sequences of bits, they may easily be transmitted between subjects.

This design alone avoids the requirement of operating system interference. If the Venti system is

provided with a read or write capability in the form of a fingerprint, it can safely allow the corresponding action. No further overhead is required to restrict access.

3 Conclusion

Venti's authors acknowledge that a fingerprint can be seen as a capability for the data to which it refers. However, they leave any more details of the system's security unexplored. We have derived additions and modifications to the Venti system that allow it to implement the common operations required by a capability-based security model.

Our final design, incorporating Venti-managed object wrappers, represents a low-overhead solution that takes advantage of the unique properties of Venti's unforgeable fingerprints.

4 Acknowledgments

The author would like to thank Professor Everett Bull and the students of CS182-1 at Pomona College in the spring of 2008. Their lectures and discussion helped introduce the author to the concepts of computer security drawn on this paper.

References

[1] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[2] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101. USENIX Association, 2002.