Key Management In Distributed Systems

Tolga Acar[1], Mira Belenkiy[1], Carl Ellison[2], Lan Nguyen[1]

[1] Extreme Computing Group, Microsoft Research, Microsoft
One Microsoft Way, Redmond, WA 98052
{tolga,mibelenk,languyen}@microsoft.com
[2] cme@acm.org

1 Abstract

We developed a cryptographic key management system for distributed networks. Our system handles every aspect of key management, including the key lifecycle, key distribution, access control, and cryptographic algorithm agility. Our software client accesses keys and other metadata stored in a distributed repository. Our system hides all key management tasks from the user; the user specifies a key management policy and our system enforces this policy. Clients perform key management tasks whenever the end user accesses keys to protect or retrieve data; there are no scheduled processes or network listeners. The repository does not need to perform any additional tasks beyond its normal course of operation: storing, servicing, and replicating data. While our system can work with a generic repository, our repository implementation is based on Microsoft Active Directory. Our system prevents data loss even if the underlying repository does not ensure consistency/atomic operations.

2 Introduction

Securing data at rest is becoming an increasingly challenging problem for large organizations and service oriented architectures. The security industry has developed numerous protocols, such as TLS/SSL, IPsec, and SSH to protect data while it is transferred over the network [1],[2],[3],[4]. Security for stored data, especially shared stored data, is much less advanced.

Traditional models of stored data security are no longer sufficient. Enterprise IT departments used to assume that data on the company intranet was secure as long as it was hidden behind firewalls and VPNs. This assumption no longer holds true due to insider threats and firewall compromises. Cloud computing and storage have a different paradigm where encryption is a good solution for providing data security. However, data encryption can only be part of the solution: the end user must still deal with distributing, securing, and renewing decryption keys. Thus, encryption simply shifts the burden from protecting data to protecting keys.

Managing cryptographic keys remains one of the hard problems in applied cryptography. Without proper key management, an otherwise theoretically secure system is in reality quite vulnerable. For example, TLS and IPsec are rendered insecure without a proper PKI [1],[2],[3], and SSH is vulnerable to man-in-the-middle attacks without trusted public keys [4],[16].

3 Architecture Overview

The distributed key management architecture presented in this paper stores keys in a distributed repository. Client software handles all key management tasks, such as key distribution, key lifecycle management, access control to keys, and cryptographic algorithm agility. Our system provides a straightforward data protection facility and relieves users from worrying about key management tasks. We created a simple data protection interface, but also allowed advanced tasks for more sophisticated users.

Our techniques can generalize to any distributed repository that provides access control facilities. We implemented our system using Microsoft Active Directory, Microsoft SQL Server, and NTFS/SMB [10],[14],[15]. This paper focuses on the key lifecycle management architecture implemented on Microsoft Active Directory, as its replication policy poses unique and interesting challenges for key management (see Section 3.1).

Our architecture does not assume any scheduled processes, additional servers (other than domain controllers), specialized software (other than clients), or network listeners beyond what is already available in a repository. Clients do not need to run any additional processes beyond what users naturally execute when they protect and retrieve data. In the case of a distributed repository, the repository also handles data replication using its own algorithms. Specifically, our implementation does not modify Active Directory schema or run any scheduled processes on the domain controllers.

3.1 Key Coherence in Distributed Systems

The **TAP Theorem** states that in an asynchronous read/write network, such as Microsoft Active Directory, it is possible to provide at most two of the following three properties: atomicity, availability, and partition tolerance [8],[9]. Fault tolerance is very desirable in a distributed system. It is important that cryptographic keys are available and are adequately protected even if some domain controllers are not available due to system failure, network partitioning, or other failures. A distributed system with built-in key lifecycle management must have the ability to survive network partitioning and rejoining. Thus, we designed our system to guarantee availability and partition tolerance. In order to do that, we had to sacrifice atomicity.

Without atomicity, we cannot guarantee that all cryptographic keys are available to all clients and all domain controllers at all times. If a client creates a cryptographic key on one domain controller, there is no guarantee that this key is available on other domain controllers within a defined time period. This can lead to two types of failures. In the worst case, two domain controllers can decide to delete or overwrite a cryptographic key while reconciling their databases; any data protected by the lost key becomes permanently unavailable. We designed our architecture to guarantee that this case does not happen. Another scenario is that a client begins to use a key that has not yet replicated to all domain controllers. In this scenario, some clients that access the “wrong” domain controller don’t have access to protected data. Our architecture tries to minimize this scenario, but, ultimately, it is impossible to prevent it due to the TAP theorem.

3.2 Key Lifecycle Management

Key lifecycle management refers to the creation and retirement of cryptographic keys. This is commonly referred to as “key rollover.” A newly generated key is often stored in the key repository along with the old keys. Since placing a key in a distributed repository is not an atomic operation, the new cryptographic key initially becomes available only to a subset of domain controllers. Depending on the repository’s replication policy, the key is eventually replicated to the remaining domain controllers over a period of time. Data protected by the new key may not be usable by all clients until the new key replicates throughout the repository. For example, this may happen if the protected data is stored in a highly available cloud storage that is independent of the key repository. Our system activates a new key (starts using it) only sometime after it is created in order to mitigate this problem.

This paper considers age-based key lifecycles. Another paradigm is to base the lifecycle around the key usage count (or some other monotonically increasing counter). However, maintaining a monotonically increasing counter is difficult in a distributed system. This becomes even more challenging if we want to leverage existing distributed repositories while making as few changes as possible. Thus, we chose an age based approach to be the most appropriate for our system.

Our system manages the key lifecycle transparently to end users. The client creates, activates, and retires keys automatically when the end user protects and retrieves data. The client’s ability to do this depends on the user’s access privileges. We give the end user the option of configuring some aspects of the key lifecycle policy, but the system takes responsibility for enforcing this policy without help from the user.

3.3 Access Control

All key management systems must limit access to cryptographic keys to authorized users. We can delegate access control management to the repository. The system presents end users with an intuitive interface for making access control policy assignments, such as granting access to a particular security group to a group of keys. Active Directory groups data into containers. Each container and each object can have its own access control list that states which users have read or write privileges. Our system stores keys belonging to each group in a different container.

3.4 Cryptographic Algorithm Agility

All cryptographic algorithms eventually become obsolete. Even in the case of public-key cryptosystems, it is eventually necessary to increase the length of the key. Our system stores meta-data along with cryptographic keys inside the repository in a way that allows the end user to configure the cryptographic policy.

Section 4 describes our key lifecycle model, Section 5 gives an in-depth description of a generic key distribution topology, and Section 6 describes our key management architecture, which uses Microsoft Active Directory as the key repository.

4 Key Lifecycle Model

We identify seven states in the lifecycle of a cryptographic key. Most readers are already familiar with some of these phases, and they resemble the phases identified in an Information Lifecycle Management (ILM) system [12]. We identified several fine grained states in a key's lifetime that affect the inner workings of a distributed key management system.

We define the six states a key goes through in its lifecycle.

1. **Creation**. A key object is created on at least one domain controller, but its attributes (such as key value) are not set.
2. **Initialization**. The key object has all its core key attributes set on at least one domain controller.
3. **Full Distribution**. An initialized key is available to all domain controllers.
4. **Active**. An initialized key is available for all cryptographic operations on at least one domain controller.
5. **Inactive**. An initialized key is unavailable for some cryptographic operations on all domain controllers.
6. **Termination**. An initialized key is permanently deleted from all domain controllers.

As soon as we begin to talk about states, we must consider transitions between these states. Transitions are events and triggers that move a cryptographic key from one state to another. Figure 1 provides a generic key lifecycle state transition diagram, while the ensuing subsections give a detailed description of each state and transitions.

It would be naïve to assume that there is only one active cryptographic key across an entire distributed system. Even if we ignore keys created for different purposes or subsets of users, managing multiple keys is still a problem. Since we cannot guarantee atomicity, we must accept that network fragmentation, replication lags, and other failures will lead to multiple active keys. A distributed key management system must be able to deal with these keys in a coherent manner.

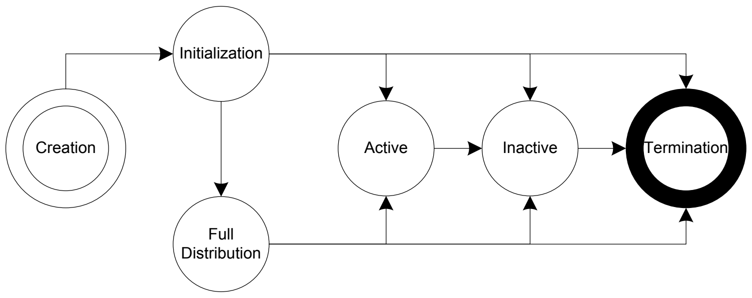

Figure 1. Cryptographic key lifecycle state diagram for a distributed system.

4.1 Creation

We say that a key has been created once a key object first appears somewhere in the key repository. However, a key in this state is not usable until it goes through the initialization state. During creation, the key object is stored in the repository with place-holders for its attributes, such as key value, key creation timestamp, and key type and format.

We assume the ability to create an object on a single domain controller in a single atomic operation. That is, no replication takes place before successful object creation, and a failure in object creation leaves no object residue on the domain controller: no partial objects. The requirement guarantees a successful transition into the initialized state.

A key object may be created and replicated across multiple domain controllers before it is initialized. In this case, we refer to the replicated keys as instances of the same created key object. Hence, there is still just one key (or key object), and it happens to exist in the created state on multiple domain controllers.

4.2 Initialization

A key object is considered initialized once it has values for all the core key attributes (defined below) on at least one domain controller. A key object is partially initialized if it does not have all of the core key attributes set. We consider a partially initialized key object to be in the creation state. If the process of writing key attributes is not atomic, a partially initialized key object may be replicated. A partially initialized and replicated key is in the created state until at least one replica is fully initialized.

There are many attributes that can be associated with a key, but we are only interested in a minimal set that we call the *core key attributes*.

Definition: Core key attributes. This is the minimal set of cryptographic key attributes that must be present for a cryptographic key to be usable by the appropriate cryptographic algorithm in an age-based key lifecycle management system. We list the three core key attributes.

- **Value.** This can include both the secret and public parts of the key. A key object without a value is useless; every keyed cryptographic operation requires a key value.
- **Creation Time.** An age-based key lifecycle management system needs to know when a key was created.
- **Key Type.** The system stores meta-data indicating how to parse and use the key, such as intended algorithm(s), bit-order, and other parsing information.

In repositories with atomic object creation and attribute population capabilities, the creation and initialization states are equivalent. We treat these states as distinct because we want to be as general as possible.

In the initialized state, some domain controllers may have no copy of the key, some domain controllers may have a copy of an uninitialized (created) key, and at least one domain controller has a copy of the initialized key object. There must be at least one domain controller that does not have a copy of the initialized key object. Otherwise, we would consider this to be the fully distributed state. An initialized key object is usable and can safely transition to the active state.

We conclude with a caveat. A key management system must gracefully handle initialized states where two or more domain controllers have the same key object initialized to different attributes values. Depending on the replication policy in a distributed repository, one of the instances of the key object may be overwritten. If the overwritten key has previously been used to protect data, this data would be irretrievably lost. The key management system we present in Section 6 guarantees that the same key object is never initialized to different values on two different domain controllers.

4.3 Full Distribution

Once an initialized key object is available on all repository instances, we say that it has transitioned to the fully distributed state. In situations where there is only one domain controller or instantaneous (atomic) replication, the initialized and fully distributed state are equivalent.

In a fragmented network, full replication may take a long time or may never happen. A key may transition into the inactive state before full distribution takes place. This happens when the key's lifetime is shorter than the time it takes to achieve full distribution.

4.4 Active

An active key is available for use in cryptographic operations on at least one domain controller. A key can become active even though it is not replicated to all domain controllers. Hence, there can be a transition directly from the initialized state to the active state. Even if the key is marked inactive or removed entirely from some domain controllers, it is still active as long as it is available on at least one domain controller.

4.5 Inactive

A key enters the inactive state when it is not considered active by any client or domain controller. It remains in the inactive state until it is removed from every domain controller. An inactive key can only be used to decrypt or verify previously generated cryptographic data.

From a cryptographic point of view, cryptographic keys have a limited life. However, in practical applications, it may be necessary to keep keys indefinitely. Old keys are needed to decrypt data or verify the signatures. The inactive state indicates a retired key; it can be used to perform cryptographic operations such as decryption and verification, but is no longer valid for others such as encryption and signing.

4.6 Termination

A terminated key has been permanently deleted in an unrecoverable fashion. A key is not terminated until it has been removed from all domain controllers. Similar to partial key distribution, a key can be deleted from some domain controllers but not others. To avoid data loss, it is crucial that key is not terminated until it has transitioned to the inactive state and all data encrypted with the key has been either decrypted or re-encrypted with another key.

5 Our Model

Our architecture contains three types of actors: end users, software clients, and domain controllers.   Following Active Directory terminology, we use the term *domain controller* to refer to a node in a distributed repository [10]. A client is a piece of software that runs on any machine that needs access to the cryptographic keys in the repository. Finally, the end user is a human being (or a surrogate process for a human being) that uses the client to access cryptographic keys.

5.1 Network Issues

A client may access one or more domain controllers at a time; however, any domain controller may become temporarily or permanently unavailable. We assume that a client can perform a single read/write operation atomically on a single domain controller. For example, if the client writes a 128-bit key value into a key object, the client is guaranteed that either all or none of the 128 bits are written.

Domain controllers may have a network independent of clients' network. Client A can encrypt and send data to Client B before Client A's domain controller has a chance to replicate the decryption key to Client B's domain controller. For example, Client A and Client B could both use the internet to store and retrieve encrypted data from an untrusted Cloud service; meanwhile, the repository functions over a corporate intranet that is temporarily fragmented.

We assume that all network connections between clients and domain controllers are mutually authenticated, and provide confidentiality and integrity. Typical examples of such connections are Kerberos and mutual SSL [5],[6].

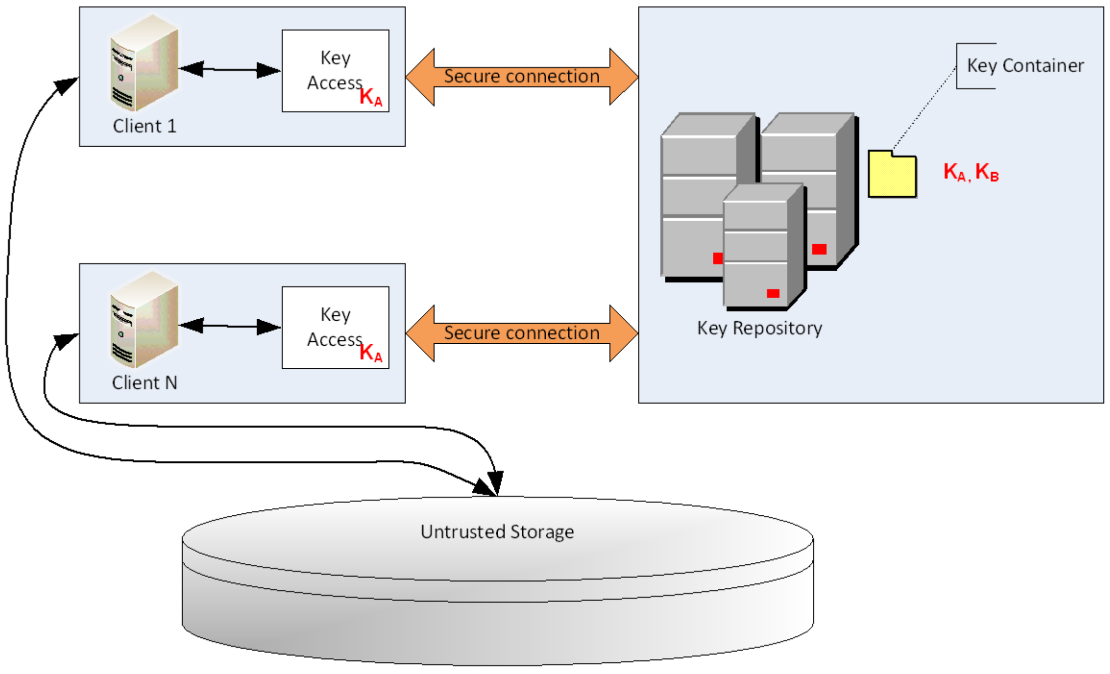

Figure 2. Distributed Key Environment.

Figure 2 depicts a key management system for a distributed network that uses a distributed repository to store cryptographic keys. Clients access any domain controller over a secure connection. There is an untrusted data store where clients store and retrieve encrypted data. A network fragmentation in the key repository does not translate to network fragmentation among clients.

5.2 Key Coherence

A key management system must address "key coherence" problems in a distributed key repository. It is necessary to minimize decryption failures due to key unavailability. Key unavailability occurs when a key is not yet replicated to a domain controller.

Theorem. To ensure all clients can decrypt all protected data at all times, a cryptographic key cannot become active until after it is fully distributed.

Proof: Figure 3 shows a mini-topology. The top two rectangles represent domain controllers, while the bottom two rectangles represent clients. Client 1 uses a key that is created on DC1 to encrypt data. Client 1 then sends the data to Client 2. Then Client 2 tries to decrypt the data by retrieving the decryption key from DC2. Meanwhile, DC1 replicates the key to DC2. We represent dataflow time delays using triangles angled in the direction of the dataflow.

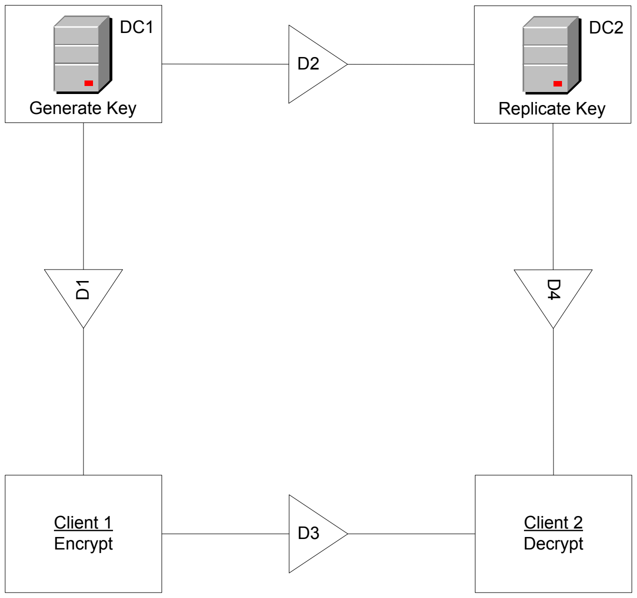

Figure 3. Key update processing nodes and delays.

> **D1**. This is the time between when the key becomes active in DC1 and when Client 1 uses it to encrypt data.
> **D2**. This is the replication lag between DC1 and DC2. We assume a constant delay to simplify analysis.
> **D3**. This is how long it takes to transmit encrypted data from Client 1 to Client 2 (possibly via a mutually accessible external storage device).
> **D4**. This is how long it takes for Client 2 to receive a key from DC2.

To ensure that Client 2 can always decrypt data it receives from Client 1, we must ensure that:

$$\text{Max}(D1, D2) <= D1 + D3 + D4$$

In the worst case, there is no lag between when Client 1 encrypts and when Client 2 tries to decrypt: D3+D4=0. This means that D2 <= D1. Thus, to ensure all encrypted data is available, a key must be fully distributed before it becomes active. □

In real life, there is no guaranteed upper bound on D2. However, it is often possible to estimate a realistic upper bound for a particular system. (In the case of extreme network fragmentation, such as off-shore oil rigs or nuclear submarines, D3 is likely to equal D2). Our observations indicate that D2= 7 days is a good, and probably an overly safe, estimate for most Microsoft Active Directory deployments.

6 Key Management Using Active Directory

We created a software client that enables users on multiple machines to securely share data. The client provides an interface for protecting data (to ensure confidentiality and integrity) and to recover the data (decrypt and verify). We call our system the Distributed Key Manager, DKM for short.

Our data protection mechanism is an extension of the existing Windows Data Protection application programming interface (DPAPI) [11]. DKM removes the single-user and single-machine requirement from DPAPI. DKM also offers cryptographic agility, letting the end user specify which algorithms to use to protect the data.

6.1 Microsoft Active Directory

Microsoft Active Directory is a distributed repository [10]. We focus on two object classes: containers and objects. Containers can hold multiple objects and containers or be empty. Both containers and objects can have attributes. Some are standard to all objects and containers, while others depend on the specific object and container class.

Each object and container can be uniquely identified in two ways. The primary method of identification is a unique GUID. When a client creates an object or container in Active Directory, the domain controller that processes the request assigns a randomly chosen GUID. In addition, each object and container has a name attribute; the root to object path also serves to uniquely identify the object.

As mentioned in Section 3.1, due to the TAP theorem, any distributed system can provide at most two of the following three properties: (1) Consistency, (2) Availability, and (3) Tolerance to network Partitions [8],[9]. Active Directory picked availability and tolerance to network partitions, sacrificing consistency. In case of a network partition, a domain controller will perform data replication within its reachable subnetwork, resulting in inconsistent states across the domain. Inconsistencies can occur even without network partition due to replication delays.

Domain controllers use a **last-write wins** policy during data replication/reconciliation. In the simplest case, if an object with the same GUID exists on DC1 and DC2, then the two domain controllers will both use the most recently modified version of the object. Deletion counts as a “write,” hence, if an object is deleted from DC1, then during replication, DC2 will also delete the object. On the other hand, if an object hasn’t been deleted from DC1, this means that this is a newly created object and it will be replicated to DC1.

Active Directory lets us perform several operations atomically on a domain controller. We can create an object as a single atomic operation. We can also write all object attributes at once atomically. However, Active Directory does not make any atomic guarantees between domain controllers.

Domain controllers perform access control via Access Control Lists (ACLs). Each Access Control Entry (ACE) in the ACL contains the unique identity of a security principal, a set of actions (Read, Write, Delete, etc), and whether those actions are permitted or denied. Each object or container can choose whether it inherits the ACLs from its parent container. Even if the object or container inherits permissions, it can still add its own ACEs to its inherited ACL.

An Active Directory security principal can be either a user or a group of users. It can also correspond to machine accounts, or general network service accounts. Security principals can be nested: a security principal can contain one or more security principals.

6.2 Key Storage and Key Distribution

Clients store and retrieve cryptographic keys from an Active Directory domain controller. By default, the client uses its machine's preferred domain controller (or whichever one happens to be available). Figure 4 shows how we store cryptographic keys in Active Directory.

We created a root container called "Distributed KeyMan" to hold all keys and key related data. This container can be placed anywhere in the Active Directory structure. In order to provide coherent access control, we stop ACL inheritance and removed some default Windows security principals from the access control list.

The Distributed KeyMan container holds one or more Service Containers (ACLed to different users). Each Service Container contains a special object called the Crypto Policy. The two important attributes in the Crypto Policy are the current active key and the maximum key lifetime. A key should become Inactive when the elapsed time since its creation exceeds the key lifetime. We discuss this more in the section on our key update algorithm.

The Crypto Policy also contains information about the intended use of the cryptographic keys: key length, crypto algorithm, etc. A client creates and initializes a new key based on the crypto policy. We also have a default Crypto Policy in the Distributed KeyMan container; if a Service Container does not have a Crypto Policy, the client assumes that it inherits the default Crypto Policy.

Keys contain the core attributes of value, creation time, and key type. Active Directory automatically assigns all objects a creation time. The client generates a random number and sets it as the key value. The client also copies the ISO algorithm ID (also called an OID) from the Crypto Policy into the key object.

We use container and contact class objects from the default Active Directory schema to avoid extending the schema.

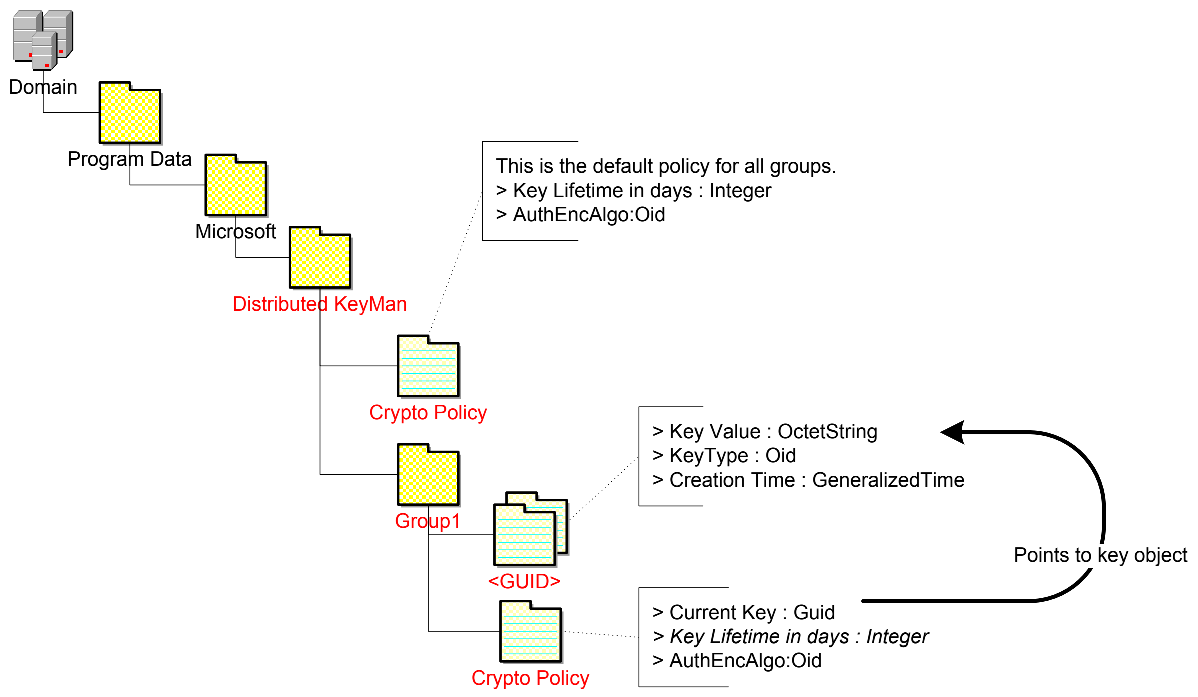

Figure 4. Storing Keys in Active Directory.

6.3 Access Control

We leverage the access control mechanism in Active Directory to control access to cryptographic keys. Users create Service Containers to hold keys under the Distributed KeyMan container. The creator of the Service Container can set the ACL on the Service Container; the keys and crypto policy inside the Service Container inherit the ACL.

In a simple use case, each end user would create his own Service Container and ACL it to his own machines/service accounts. In a more complex scenario, a single service would require multiple Service Containers. Our design allows users to think of access control from a functional perspective rather than a per-key perspective.

6.4 Key Lifecycle Management

The most important rule for our key management strategy is that a client must never modify a key that it did not create. The heart of our key lifecycle management system is our key update algorithm with the following characteristics:

1. Clients in isolated subnetworks can safely update keys on their local domain controllers.
2. When domain controllers on isolated subnetworks reconnect with the rest of the repository, updated keys are merged automatically as part of the standard Active Directory replication/reconciliation process.
3. Users do not have to worry about the key update algorithm, schedule, or mechanism.
4. Clients do not have to coordinate their actions with other clients. Each client can safely run the key update algorithm on its local domain controller without worrying that a different client has modified the same Service Container on a different domain controller.
5. Key update is performed gradually, over time, increasing the chance of full distribution before a key becomes Active.
6. The key update algorithm is tolerant to failure at all points in the algorithm and guarantees no data loss.

Despite our best efforts, there is a miniscule chance that a key object can become initialized to two different values on two different domain controllers. This can occur if a client creates a key object and its local domain controller happens to assign it the same GUID as a key object on another domain controller. In this case, even though each client initializes only the key object it created, the two domain controllers will treat both key objects as instances of the same key object. One of the key objects could be overwritten during replication. If the overwritten key object had previously been used to protect data, this would result in permanent irrecoverable data loss. Fortunately, Active Directory uses 128-bit GUIDs, and due to the Birthday Paradox, the probability of two domain controllers choosing the same GUID is on the order of 2^{-64}, which is negligible.

General Approach

A DKM client creates and initializes a new key object whenever the Service Container's current active key approaches expiration. This happens anytime a user tries to protect data within seven days of the key expiring. The client first checks that someone else hasn't already created a new key. The first time a client needs to protect data after the current key has expired, the client sets the most recently created key object in the Service Container as the current key in the Crypto Policy. See Figure 5 for our full key update algorithm.

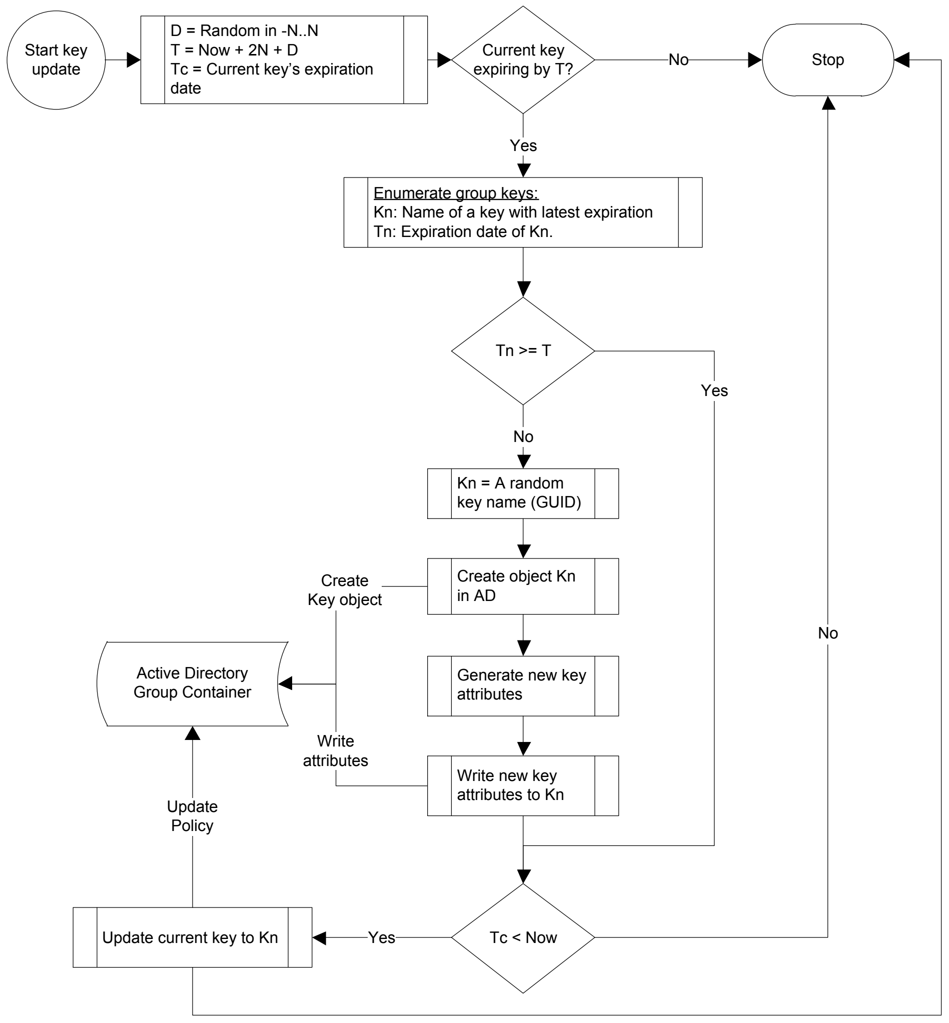
Start key update
D = Random in -N..N
T = Now + 2N + D
Tc = Current key's expiration date
Current key expiring by T?
No
Stop
Yes
Enumerate group keys:
Kn: Name of a key with latest expiration
Tn: Expiration date of Kn.
Tn >= T
Yes
No
Kn = A random key name (GUID)
Create object Kn in AD
Create Key object
Active Directory Group Container
Generate new key attributes
No
Write attributes
Write new key attributes to Kn
Update Policy
Update current key to Kn
Yes
Tc < Now

Figure 5. Key update algorithm. N=7 (1 week).

We made a conscious decision to perform key update tasks only when the user protects data. During data recovery (decryption and verification), the client does not need the current active key – it only needs to access the original data protection key for that ciphertext.

6.5 State Transitions

DKM clients perform key lifecycle management tasks on a domain controller and let Active Directory handle replication. Unless otherwise stated, the client performs the following actions:

1. **Creation**. The client checks the Crypto Policy in the Service Container to see if the current Active key expires within the next seven days. If so, and if the Service Container does not have any key objects that have been created more recently than the current Active key, the client creates a blank key object. The domain controller automatically sets the GUID and creation timestamp.
2. **Initialization**. Immediately after creating a key object, the client writes all the key attributes: key value, algorithm OIDs, and key lifetime. If the key object replicates to another domain controller before initialization, subsequent replication will fill-in the key attributes. There is a slight possibility that the

client's domain controller will go off-line between key creation and key initialization; we discuss how DKM recovers from this later. The Active Directory last-write-wins policy ensures that on each domain controller, instances of the same key object will either be in the created state or initialized to the same value.

3. **Full Distribution**. Depending on replication schedules, network delay, and network fragmentation, keys may or may not reach this state.
4. **Active**. If a client sees that the current key specified in the Crypto Policy has expired, the client will need to use a different key. The client checks whether the most recent key that exists on the domain controller is in the initialized state. If so, the client modifies the Crypto Policy to make the most recent key the current Active key. If the most recent key has not been initialized, the client creates a new key object, initializes it, and modifies the Crypto Policy to point to the new key as the current Active key.
5. **Inactive**. A client modifies the Crypto Policy on a domain controller to use a different key as the current active key. When this change replicates to all domain controllers, the key becomes Inactive.
6. **Terminated**. A user has to explicitly ask a client to delete a key. The client will look at the state of the Service Container on the domain controller, and if necessary create a new key object, initialize it, and set it as the current active key. Then the client will delete the key.

6.6 Fault Tolerance

Our key update algorithm is resilient to key coherence problems that can arise when the network becomes partitioned. Clients that receive protected data out-of-band will be temporarily unable to decrypt the data. We have to accept this failure as an inevitable problem in distributed systems with network partitions and no coherence guarantee.

Suppose a client creates a key object on a domain controller and then that domain controller goes off-line before the client can initialize the key. The client will swallow the key update error, and, if it is unable to get the current Active key to perform its data protection task, give the end user a data protection error. When the domain controller rejoins the network, the created key will replicate to other domain controllers. However, it will never become initialized, and therefore, never enter the Active state.

If, for some reason a client detects that the current Active key is uninitialized (or in some other invalid state), it creates a new key and sets that as the current Active key. It is important that a client never modifies a key that it did not create.

When a client detects that the Crypto Policy has a missing or invalid current key identifier, the client invokes our self-healing mechanism. The client retrieves all the keys from the Service Container, and picks the most recently generated key. It tries to update the Crypto Policy to use this key. Even if the client encounters an error (usually because it does not have permission to modify the Crypto Policy), it still uses the most recent key as if it was the current Active key.

6.7 Data Loss

DKM guarantees that there will never be any data loss. Protected data might be unavailable temporarily due to network partitions. However, our key update algorithms are designed to ensure that any key that has ever been Active will eventually replicate to all domain controllers. DKM clients achieve this by following one simple rule: never modify a key object that the DKM client did not itself create. Any key management problem is resolved by creating a new key or modifying the Crypto Policy.

There is only one scenario when a user might lose data. If the user chooses to terminate a key while it is still active on one or more domain controllers, then the user risks losing the ability to recover protected data. Since there is no way to guarantee that a particular key is Inactive on all domain controllers (indeed, a single domain controller might be down for repairs, and then later return with the key in the Active state), DKM clients never delete keys on their own. This action must be explicitly taken by the user.

6.8 Optimization

DKM is designed for users who might need to frequently protect data. One of our scenarios is to support a room full of servers, each of which makes over a thousand data protection calls per minute. During the seven days before a key expires, there is a danger of each server creating millions of new keys in a Service Container. Therefore, DKM clients check whether an initialized key exists before creating a new one. In addition, the DKM client randomly chooses when in the seven day period it will create a new key – it picks an integer $T \in [1,7]$, and runs key update only if the current date is less than T days from the expiration date.

Our architecture employs client-side caching in order to reduce network bandwidth usage, increase tolerance to network outages and network transmission delays, and to improve response time to data protection requests. There are two caches: a key cache and a Crypto Policy cache. Recognizing that keys are not deleted in the normal course of operation, there is no need to remove a key from the cache except to prevent unlimited cache expansion. DKM clients retrieve keys from the repository on demand, and store them in the client-side memory cache. Cached keys are encrypted in memory with a local key (provided by the operating system) to limit their exposure in page swap files and crash dumps.

7 Conclusion

This paper proposes a key lifecycle management architecture for distributed environments that can be used for cryptographic data protection. The implementation described in this paper uses Microsoft Active Directory as the distributed repository for storing keys and cryptography policies. We present a novel solution to addresses a unique combination of challenges inherent in cryptographic key management and Active Directory itself.

Our system allows users to specify (and modify) all aspects of the key management policy, regarding cryptography, key rollover, and access control. Policy enforcement is handled entirely by software clients in a way that is transparent to the user. There are no scheduled processes; instead, users trigger the client to execute key management tasks in the normal course of data protection and retrieval.

Our key update algorithm minimizes the chance of protected data becoming unavailable to some clients due to discrepancies in the data distribution and repository replication networks. Our key update algorithm is also guaranteed prevent data loss due to Active Directory's replication policy of last-write-wins. Finally, the algorithm is optimized to limit the number of extra keys that clients create.

Using Active Directory as our key repository allowed us to leverage its extensive access control features. Users can designate that different documents belong to different groups by creating multiple service containers. Active Directory domain controllers ensure that only group members can access keys stored in their service container. To simplify things even further for the user, our clients let users specify which key management tasks they are willing to entrust to other users (passively using keys to access data, updating keys, managing access control) and then the client sets the appropriate ACLs.

In conclusion, we present a simple, practical key management architecture that works with existing distributed repositories.

8 References

1. RFC 5246, "The Transport Layer Security (TLS) Protocol Version 1.2", http://www.ietf.org/rfc/rfc5246.txt.
2. RFC 2409, "The Internet Key Exchange (IKE)", http://www.ietf.org/rfc/rfc2409.txt.
3. RFC 4306, "The Internet Key Exchange (IKEv2)", http://www.ietf.org/rfc/rfc4306.txt.
4. RFC 4251, "The Secure Shell (SSH) Protocol Architecture", http://www.ietf.org/rfc/rfc4251.txt.

5. [MS-KILE] Microsoft Corporation, "Kerberos Protocol Extensions", http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5bMS-KILE%5d.pdf, January 2007.
6. RFC4120, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005, http://www.ietf.org/rfc/rfc4120.txt.
7. Electronic Key Management System (EKMS), http://en.wikipedia.org/wiki/EKMS
8. Eric A. Brewer, Towards Robust Distributed Systems (Invited Talk), Principles of Distributed Computing, Portland, Oregon, July 2000.
9. S.Gilbert, N.Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, Sigact News, Vol.33, pp.51-59, 2002.
10. Active Directory Domain Services, http://msdn.microsoft.com/en-us/library/aa362244(VS.85).aspx
11. Windows Data Protection, http://msdn.microsoft.com/en-us/library/ms995355.aspx
12. Petrocelli, T., "Data Protection and Information Lifecycle Management", 2005, Prentice hall.
13. RFC 5247, "Extensible Authentication Protocol (EAP) Key Management Framework", http://tools.ietf.org/html/rfc5247, August 2008.
14. Microsoft SMB Protocol and CIFS Protocol Overview, http://msdn.microsoft.com/en-us/library/aa365233(VS.85).aspx
15. NTFS Technical Reference, http://technet.microsoft.com/en-us/library/cc758691(WS.10).aspx, March 2003.
16. M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D.A. Osvik, B. de Weger, "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate", Crypto 2009, Santa Barbara, CA, August 16-20, 2009