author	Eric Biggers <ebiggers@google.com>	2022-11-04 15:54:38 -0700
committer	Greg Kroah-Hartman <gregkh@linuxfoundation.org>	2022-11-10 18:17:30 +0100
commit	68d15d6558a386f46d815a6ac39edecad713a1bf (patch)	
tree	3009422a6f187f853cfd182d642c9e824a57e413	
parent	e1aada9b71493b2e11c2a239ece99a97e3f13431 (diff)	
download	linux-68d15d6558a386f46d815a6ac39edecad713a1bf.tar.gz	

fscrypt: stop using keyrings subsystem for fscrypt_master_key

commit d7e7b9af104c7b389a0c21eb26532511bce4b510 upstream.

The approach of fs/crypto/ internally managing the fscrypt_master_key structs as the payloads of "struct key" objects contained in a "struct key" keyring has outlived its usefulness. The original idea was to simplify the code by reusing code from the keyrings subsystem. However, several issues have arisen that can't easily be resolved:

- When a master key struct is destroyed, blk_crypto_evict_key() must be called on any per-mode keys embedded in it. (This started being the case when inline encryption support was added.) Yet, the keyrings subsystem can arbitrarily delay the destruction of keys, even past the time the filesystem was unmounted. Therefore, currently there is no easy way to call blk_crypto_evict_key() when a master key is destroyed. Currently, this is worked around by holding an extra reference to the filesystem's request_queue(s). But it was overlooked that the request_queue reference is *not* guaranteed to pin the corresponding blk_crypto_profile too; for device-mapper devices that support inline crypto, it doesn't. This can cause a use-after-free.

- When the last inode that was using an incompletely-removed master key is evicted, the master key removal is completed by removing the key struct from the keyring. Currently this is done via key_invalidate(). Yet, key_invalidate() takes the key semaphore. This can deadlock when called from the shrinker, since in fscrypt_ioctl_add_key(), memory is allocated with GFP_KERNEL under the same semaphore.

- More generally, the fact that the keyrings subsystem can arbitrarily delay the destruction of keys (via garbage collection delay, or via random processes getting temporary key references) is undesirable, as it means we can't strictly guarantee that all secrets are ever wiped.

- Doing the master key lookups via the keyrings subsystem results in the key_permission LSM hook being called. fscrypt doesn't want this, as all access control for encrypted files is designed to happen via the files themselves, like any other files. The workaround which SELinux users are using is to change their SELinux policy to grant key search access to all domains. This works, but it is an odd extra step that shouldn't really have to be done.

The fix for all these issues is to change the implementation to what I should have done originally: don't use the keyrings subsystem to keep track of the filesystem's fscrypt_master_key structs. Instead, just store them in a regular kernel data structure, and rework the reference counting, locking, and lifetime accordingly. Retain support for

RCU-mode key lookups by using a hash table. Replace fscrypt_sb_free()
with fscrypt_sb_delete(), which releases the keys synchronously and runs
a bit earlier during unmount, so that block devices are still available.

A side effect of this patch is that neither the master keys themselves
nor the filesystem keyrings will be listed in /proc/keys anymore.
("Master key users" and the master key users keyrings will still be
listed.) However, this was mostly an implementation detail, and it was
intended just for debugging purposes. I don't know of anyone using it.

This patch does *not* change how "master key users" (->mk_users) works;
that still uses the keyrings subsystem. That is still needed for key
quotas, and changing that isn't necessary to solve the issues listed
above. If we decide to change that too, it would be a separate patch.

I've marked this as fixing the original commit that added the fscrypt
keyring, but as noted above the most important issue that this patch
fixes wasn't introduced until the addition of inline encryption support.

Fixes: 22d94f493bfb ("fscrypt: add FS_IOC_ADD_ENCRYPTION_KEY ioctl")
Signed-off-by: Eric Biggers <ebiggers@google.com>
Link: https://lore.kernel.org/r/20220901193208.138056-2-ebiggers@kernel.org
Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>

Diffstat

-rw-r--r--	fs/crypto/fscrypt_private.h	71
-rw-r--r--	fs/crypto/hooks.c	10
-rw-r--r--	fs/crypto/keyring.c	486
-rw-r--r--	fs/crypto/keysetup.c	81
-rw-r--r--	fs/crypto/policy.c	8
-rw-r--r--	fs/super.c	2
-rw-r--r--	include/linux/fs.h	2
-rw-r--r--	include/linux/fscrypt.h	4

8 files changed, 353 insertions, 311 deletions

```
diff --git a/fs/crypto/fscrypt_private.h b/fs/crypto/fscrypt_private.h
index 3afdaa0847736b..577cae7facb013 100644
--- a/fs/crypto/fscrypt_private.h
+++ b/fs/crypto/fscrypt_private.h
@@ -225,7 +225,7 @@ struct fscrypt_info {
          * will be NULL if the master key was found in a process-subscribed
          * keyring rather than in the filesystem-level keyring.
          */
-        struct key *ci_master_key;
+        struct fscrypt_master_key *ci_master_key;

         /*
          * Link in list of inodes that were unlocked with the master key.
@@ -437,6 +437,40 @@ struct fscrypt_master_key_secret {
 struct fscrypt_master_key {

         /*
+         * Back-pointer to the super_block of the filesystem to which this
+         * master key has been added.  Only valid if ->mk_active_refs > 0.
+         */
+        struct super_block                      *mk_sb;
+
+        /*
+         * Link in ->mk_sb->s_master_keys->key_hashtable.
+         * Only valid if ->mk_active_refs > 0.
+         */
+        struct hlist_node                       mk_node;
+
+        /* Semaphore that protects ->mk_secret and ->mk_users */
```

```
+       struct rw_semaphore                     mk_sem;
+
+       /*
+        * Active and structural reference counts.  An active ref guarantees
+        * that the struct continues to exist, continues to be in the keyring
+        * ->mk_sb->s_master_keys, and that any embedded subkeys (e.g.
+        * ->mk_direct_keys) that have been prepared continue to exist.
+        * A structural ref only guarantees that the struct continues to exist.
+        *
+        * There is one active ref associated with ->mk_secret being present,
+        * and one active ref for each inode in ->mk_decrypted_inodes.
+        *
+        * There is one structural ref associated with the active refcount being
+        * nonzero.  Finding a key in the keyring also takes a structural ref,
+        * which is then held temporarily while the key is operated on.
+        */
+       refcount_t                              mk_active_refs;
+       refcount_t                              mk_struct_refs;
+
+       struct rcu_head                         mk_rcu_head;
+
+       /*
         * The secret key material.  After FS_IOC_REMOVE_ENCRYPTION_KEY is
         * executed, this is wiped and no new inodes can be unlocked with this
         * key; however, there may still be inodes in ->mk_decrypted_inodes
@@ -444,7 +478,10 @@ struct fscrypt_master_key {
         * FS_IOC_REMOVE_ENCRYPTION_KEY can be retried, or
         * FS_IOC_ADD_ENCRYPTION_KEY can add the secret again.
         *
-        * Locking: protected by this master key's key->sem.
+        * While ->mk_secret is present, one ref in ->mk_active_refs is held.
+        *
+        * Locking: protected by ->mk_sem.  The manipulation of ->mk_active_refs
+        *          associated with this field is protected by ->mk_sem as well.
         */
        struct fscrypt_master_key_secret        mk_secret;

@@ -465,23 +502,13 @@ struct fscrypt_master_key {
        *
        * This is NULL for v1 policy keys; those can only be added by root.
        *
-        * Locking: in addition to this keyring's own semaphore, this is
-        * protected by this master key's key->sem, so we can do atomic
-        * search+insert.  It can also be searched without taking any locks, but
-        * in that case the returned key may have already been removed.
+        * Locking: protected by ->mk_sem.  (We don't just rely on the keyrings
+        * subsystem semaphore ->mk_users->sem, as we need support for atomic
+        * search+insert along with proper synchronization with ->mk_secret.)
        */
       struct key              *mk_users;

       /*
-        * Length of ->mk_decrypted_inodes, plus one if mk_secret is present.
-        * Once this goes to 0, the master key is removed from ->s_master_keys.
-        * The 'struct fscrypt_master_key' will continue to live as long as the
-        * 'struct key' whose payload it is, but we won't let this reference
-        * count rise again.
-        */
-       refcount_t              mk_refcount;
-
-       /*
        * List of inodes that were unlocked using this key.  This allows the
        * inodes to be evicted efficiently if the key is removed.
        */
@@ -506,10 +533,10 @@ static inline bool
```

```
 is_master_key_secret_present(const struct fscrypt_master_key_secret *secret)
 {
        /*
-        * The READ_ONCE() is only necessary for fscrypt_drop_inode() and
-        * fscrypt_key_describe().  These run in atomic context, so they can't
-        * take the key semaphore and thus 'secret' can change concurrently
-        * which would be a data race.  But they only need to know whether the
+        * The READ_ONCE() is only necessary for fscrypt_drop_inode().
+        * fscrypt_drop_inode() runs in atomic context, so it can't take the key
+        * semaphore and thus 'secret' can change concurrently which would be a
+        * data race.  But fscrypt_drop_inode() only need to know whether the
         * secret *was* present at the time of check, so READ_ONCE() suffices.
         */
        return READ_ONCE(secret->size) != 0;
@@ -538,7 +565,11 @@ static inline int master_key_spec_len(const struct fscrypt_key_specifier *spec)
        return 0;
 }

-struct key *
+void fscrypt_put_master_key(struct fscrypt_master_key *mk);
+
+void fscrypt_put_master_key_activeref(struct fscrypt_master_key *mk);
+
+struct fscrypt_master_key *
 fscrypt_find_master_key(struct super_block *sb,
                         const struct fscrypt_key_specifier *mk_spec);


diff --git a/fs/crypto/hooks.c b/fs/crypto/hooks.c
index 7c01025879b38f..7b8c5a1104b586 100644
--- a/fs/crypto/hooks.c
+++ b/fs/crypto/hooks.c
@@ -5,8 +5,6 @@
  * Encryption hooks for higher-level filesystem operations.
  */

-#include <linux/key.h>
-
 #include "fscrypt_private.h"

 /**
@@ -142,7 +140,6 @@ int fscrypt_prepare_setflags(struct inode *inode,
                             unsigned int oldflags, unsigned int flags)
 {
        struct fscrypt_info *ci;
-       struct key *key;
        struct fscrypt_master_key *mk;
        int err;

@@ -158,14 +155,13 @@ int fscrypt_prepare_setflags(struct inode *inode,
                ci = inode->i_crypt_info;
                if (ci->ci_policy.version != FSCRYPT_POLICY_V2)
                        return -EINVAL;
-               key = ci->ci_master_key;
-               mk = key->payload.data[0];
-               down_read(&key->sem);
+               mk = ci->ci_master_key;
+               down_read(&mk->mk_sem);
                if (is_master_key_secret_present(&mk->mk_secret))
                        err = fscrypt_derive_dirhash_key(ci, mk);
                else
                        err = -ENOKEY;
-               up_read(&key->sem);
+               up_read(&mk->mk_sem);
                return err;
        }
```

```
        return 0;

diff --git a/fs/crypto/keyring.c b/fs/crypto/keyring.c
index caee9f8620dd9b..9b98d6a576e6a0 100644
--- a/fs/crypto/keyring.c
+++ b/fs/crypto/keyring.c
@@ -18,6 +18,7 @@
  * information about these ioctls.
  */

+#include <asm/unaligned.h>
 #include <crypto/skcipher.h>
 #include <linux/key-type.h>
 #include <linux/random.h>
@@ -25,6 +26,18 @@

 #include "fscrypt_private.h"

+/* The master encryption keys for a filesystem (->s_master_keys) */
+struct fscrypt_keyring {
+       /*
+        * Lock that protects ->key_hashtable.  It does *not* protect the
+        * fscrypt_master_key structs themselves.
+        */
+       spinlock_t lock;
+
+       /* Hash table that maps fscrypt_key_specifier to fscrypt_master_key */
+       struct hlist_head key_hashtable[128];
+};
+
 static void wipe_master_key_secret(struct fscrypt_master_key_secret *secret)
 {
        fscrypt_destroy_hkdf(&secret->hkdf);
@@ -38,20 +51,70 @@ static void move_master_key_secret(struct fscrypt_master_key_secret *dst,
        memzero_explicit(src, sizeof(*src));
 }

-static void free_master_key(struct fscrypt_master_key *mk)
+static void fscrypt_free_master_key(struct rcu_head *head)
+{
+       struct fscrypt_master_key *mk =
+               container_of(head, struct fscrypt_master_key, mk_rcu_head);
+       /*
+        * The master key secret and any embedded subkeys should have already
+        * been wiped when the last active reference to the fscrypt_master_key
+        * struct was dropped; doing it here would be unnecessarily late.
+        * Nevertheless, use kfree_sensitive() in case anything was missed.
+        */
+       kfree_sensitive(mk);
+}
+
+void fscrypt_put_master_key(struct fscrypt_master_key *mk)
+{
+       if (!refcount_dec_and_test(&mk->mk_struct_refs))
+               return;
+       /*
+        * No structural references left, so free ->mk_users, and also free the
+        * fscrypt_master_key struct itself after an RCU grace period ensures
+        * that concurrent keyring lookups can no longer find it.
+        */
+       WARN_ON(refcount_read(&mk->mk_active_refs) != 0);
+       key_put(mk->mk_users);
+       mk->mk_users = NULL;
+       call_rcu(&mk->mk_rcu_head, fscrypt_free_master_key);
+}
+
```

```
+void fscrypt_put_master_key_activeref(struct fscrypt_master_key *mk)
 {
+	struct super_block *sb = mk->mk_sb;
+	struct fscrypt_keyring *keyring = sb->s_master_keys;
 	size_t i;

-	wipe_master_key_secret(&mk->mk_secret);
+	if (!refcount_dec_and_test(&mk->mk_active_refs))
+		return;
+	/*
+	 * No active references left, so complete the full removal of this
+	 * fscrypt_master_key struct by removing it from the keyring and
+	 * destroying any subkeys embedded in it.
+	 */
+
+	spin_lock(&keyring->lock);
+	hlist_del_rcu(&mk->mk_node);
+	spin_unlock(&keyring->lock);
+
+	/*
+	 * ->mk_active_refs == 0 implies that ->mk_secret is not present and
+	 * that ->mk_decrypted_inodes is empty.
+	 */
+	WARN_ON(is_master_key_secret_present(&mk->mk_secret));
+	WARN_ON(!list_empty(&mk->mk_decrypted_inodes));

 	for (i = 0; i <= FSCRYPT_MODE_MAX; i++) {
 		fscrypt_destroy_prepared_key(&mk->mk_direct_keys[i]);
 		fscrypt_destroy_prepared_key(&mk->mk_iv_ino_lblk_64_keys[i]);
 		fscrypt_destroy_prepared_key(&mk->mk_iv_ino_lblk_32_keys[i]);
 	}
+	memzero_explicit(&mk->mk_ino_hash_key,
+			 sizeof(mk->mk_ino_hash_key));
+	mk->mk_ino_hash_key_initialized = false;

-	key_put(mk->mk_users);
-	kfree_sensitive(mk);
+	/* Drop the structural ref associated with the active refs. */
+	fscrypt_put_master_key(mk);
 }

 static inline bool valid_key_spec(const struct fscrypt_key_specifier *spec)
@@ -61,44 +124,6 @@ static inline bool valid_key_spec(const struct fscrypt_key_specifier *spec)
 	return master_key_spec_len(spec) != 0;
 }

-static int fscrypt_key_instantiate(struct key *key,
-				   struct key_preparsed_payload *prep)
-{
-	key->payload.data[0] = (struct fscrypt_master_key *)prep->data;
-	return 0;
-}
-
-static void fscrypt_key_destroy(struct key *key)
-{
-	free_master_key(key->payload.data[0]);
-}
-
-static void fscrypt_key_describe(const struct key *key, struct seq_file *m)
-{
-	seq_puts(m, key->description);
-
-	if (key_is_positive(key)) {
-		const struct fscrypt_master_key *mk = key->payload.data[0];
-
```

```
-               if (!is_master_key_secret_present(&mk->mk_secret))
-                       seq_puts(m, ": secret removed");
-       }
-}
-
-/*
- * Type of key in ->s_master_keys.  Each key of this type represents a master
- * key which has been added to the filesystem.  Its payload is a
- * 'struct fscrypt_master_key'.  The "." prefix in the key type name prevents
- * users from adding keys of this type via the keyrings syscalls rather than via
- * the intended method of FS_IOC_ADD_ENCRYPTION_KEY.
- */
-static struct key_type key_type_fscrypt = {
-       .name                   = "._fscrypt",
-       .instantiate            = fscrypt_key_instantiate,
-       .destroy                = fscrypt_key_destroy,
-       .describe               = fscrypt_key_describe,
-};
-
 static int fscrypt_user_key_instantiate(struct key *key,
                                        struct key_preparsed_payload *prep)
 {
@@ -131,32 +156,6 @@ static struct key_type key_type_fscrypt_user = {
        .describe               = fscrypt_user_key_describe,
 };

-/* Search ->s_master_keys or ->mk_users */
-static struct key *search_fscrypt_keyring(struct key *keyring,
-                                         struct key_type *type,
-                                         const char *description)
-{
-       /*
-        * We need to mark the keyring reference as "possessed" so that we
-        * acquire permission to search it, via the KEY_POS_SEARCH permission.
-        */
-       key_ref_t keyref = make_key_ref(keyring, true /* possessed */);
-
-       keyref = keyring_search(keyref, type, description, false);
-       if (IS_ERR(keyref)) {
-               if (PTR_ERR(keyref) == -EAGAIN || /* not found */
-                   PTR_ERR(keyref) == -EKEYREVOKED) /* recently invalidated */
-                       keyref = ERR_PTR(-ENOKEY);
-               return ERR_CAST(keyref);
-       }
-       return key_ref_to_ptr(keyref);
-}
-
-#define FSCRYPT_FS_KEYRING_DESCRIPTION_SIZE    \
-       (CONST_STRLEN("fscrypt-") + sizeof_field(struct super_block, s_id))
-
-#define FSCRYPT_MK_DESCRIPTION_SIZE    (2 * FSCRYPT_KEY_IDENTIFIER_SIZE + 1)
-
 #define FSCRYPT_MK_USERS_DESCRIPTION_SIZE      \
        (CONST_STRLEN("fscrypt-") + 2 * FSCRYPT_KEY_IDENTIFIER_SIZE + \
         CONST_STRLEN("-users") + 1)
@@ -164,21 +163,6 @@ static struct key *search_fscrypt_keyring(struct key *keyring,
 #define FSCRYPT_MK_USER_DESCRIPTION_SIZE       \
        (2 * FSCRYPT_KEY_IDENTIFIER_SIZE + CONST_STRLEN(".uid.") + 10 + 1)

-static void format_fs_keyring_description(
-                       char description[FSCRYPT_FS_KEYRING_DESCRIPTION_SIZE],
-                       const struct super_block *sb)
-{
-       sprintf(description, "fscrypt-%s", sb->s_id);
-}
```

```
-
-static void format_mk_description(
-                       char description[FSCRYPT_MK_DESCRIPTION_SIZE],
-                       const struct fscrypt_key_specifier *mk_spec)
-{
-       sprintf(description, "%*phN",
-               master_key_spec_len(mk_spec), (u8 *)&mk_spec->u);
-}
-
 static void format_mk_users_keyring_description(
                        char description[FSCRYPT_MK_USERS_DESCRIPTION_SIZE],
                        const u8 mk_identifier[FSCRYPT_KEY_IDENTIFIER_SIZE])
@@ -199,20 +183,15 @@ static void format_mk_user_description(
 /* Create ->s_master_keys if needed.  Synchronized by fscrypt_add_key_mutex. */
 static int allocate_filesystem_keyring(struct super_block *sb)
 {
-       char description[FSCRYPT_FS_KEYRING_DESCRIPTION_SIZE];
-       struct key *keyring;
+       struct fscrypt_keyring *keyring;

        if (sb->s_master_keys)
                return 0;

-       format_fs_keyring_description(description, sb);
-       keyring = keyring_alloc(description, GLOBAL_ROOT_UID, GLOBAL_ROOT_GID,
-                               current_cred(), KEY_POS_SEARCH |
-                                 KEY_USR_SEARCH | KEY_USR_READ | KEY_USR_VIEW,
-                               KEY_ALLOC_NOT_IN_QUOTA, NULL, NULL);
-       if (IS_ERR(keyring))
-               return PTR_ERR(keyring);
-
+       keyring = kzalloc(sizeof(*keyring), GFP_KERNEL);
+       if (!keyring)
+               return -ENOMEM;
+       spin_lock_init(&keyring->lock);
        /*
         * Pairs with the smp_load_acquire() in fscrypt_find_master_key().
         * I.e., here we publish ->s_master_keys with a RELEASE barrier so that
@@ -222,21 +201,75 @@ static int allocate_filesystem_keyring(struct super_block *sb)
        return 0;
 }

-void fscrypt_sb_free(struct super_block *sb)
+/*
+ * This is called at unmount time to release all encryption keys that have been
+ * added to the filesystem, along with the keyring that contains them.
+ *
+ * Note that besides clearing and freeing memory, this might need to evict keys
+ * from the keyslots of an inline crypto engine.  Therefore, this must be called
+ * while the filesystem's underlying block device(s) are still available.
+ */
+void fscrypt_sb_delete(struct super_block *sb)
 {
-       key_put(sb->s_master_keys);
+       struct fscrypt_keyring *keyring = sb->s_master_keys;
+       size_t i;
+
+       if (!keyring)
+               return;
+
+       for (i = 0; i < ARRAY_SIZE(keyring->key_hashtable); i++) {
+               struct hlist_head *bucket = &keyring->key_hashtable[i];
+               struct fscrypt_master_key *mk;
+               struct hlist_node *tmp;
+
```

```
+                hlist_for_each_entry_safe(mk, tmp, bucket, mk_node) {
+                        /*
+                         * Since all inodes were already evicted, every key
+                         * remaining in the keyring should have an empty inode
+                         * list, and should only still be in the keyring due to
+                         * the single active ref associated with ->mk_secret.
+                         * There should be no structural refs beyond the one
+                         * associated with the active ref.
+                         */
+                        WARN_ON(refcount_read(&mk->mk_active_refs) != 1);
+                        WARN_ON(refcount_read(&mk->mk_struct_refs) != 1);
+                        WARN_ON(!is_master_key_secret_present(&mk->mk_secret));
+                        wipe_master_key_secret(&mk->mk_secret);
+                        fscrypt_put_master_key_activeref(mk);
+                }
+        }
+        kfree_sensitive(keyring);
         sb->s_master_keys = NULL;
 }

+static struct hlist_head *
+fscrypt_mk_hash_bucket(struct fscrypt_keyring *keyring,
+                       const struct fscrypt_key_specifier *mk_spec)
+{
+        /*
+         * Since key specifiers should be "random" values, it is sufficient to
+         * use a trivial hash function that just takes the first several bits of
+         * the key specifier.
+         */
+        unsigned long i = get_unaligned((unsigned long *)&mk_spec->u);
+
+        return &keyring->key_hashtable[i % ARRAY_SIZE(keyring->key_hashtable)];
+}
+
 /*
- * Find the specified master key in ->s_master_keys.
- * Returns ERR_PTR(-ENOKEY) if not found.
+ * Find the specified master key struct in ->s_master_keys and take a structural
+ * ref to it.  The structural ref guarantees that the key struct continues to
+ * exist, but it does *not* guarantee that ->s_master_keys continues to contain
+ * the key struct.  The structural ref needs to be dropped by
+ * fscrypt_put_master_key().  Returns NULL if the key struct is not found.
  */
-struct key *fscrypt_find_master_key(struct super_block *sb,
-                                    const struct fscrypt_key_specifier *mk_spec)
+struct fscrypt_master_key *
+fscrypt_find_master_key(struct super_block *sb,
+                        const struct fscrypt_key_specifier *mk_spec)
 {
-        struct key *keyring;
-        char description[FSCRYPT_MK_DESCRIPTION_SIZE];
+        struct fscrypt_keyring *keyring;
+        struct hlist_head *bucket;
+        struct fscrypt_master_key *mk;

         /*
          * Pairs with the smp_store_release() in allocate_filesystem_keyring().
@@ -246,10 +279,38 @@ struct key *fscrypt_find_master_key(struct super_block *sb,
          */
         keyring = smp_load_acquire(&sb->s_master_keys);
         if (keyring == NULL)
-                return ERR_PTR(-ENOKEY); /* No keyring yet, so no keys yet. */
-
-        format_mk_description(description, mk_spec);
-        return search_fscrypt_keyring(keyring, &key_type_fscrypt, description);
```

```
+                return NULL; /* No keyring yet, so no keys yet. */
+
+        bucket = fscrypt_mk_hash_bucket(keyring, mk_spec);
+        rcu_read_lock();
+        switch (mk_spec->type) {
+        case FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR:
+                hlist_for_each_entry_rcu(mk, bucket, mk_node) {
+                        if (mk->mk_spec.type ==
+                                FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR &&
+                            memcmp(mk->mk_spec.u.descriptor,
+                                   mk_spec->u.descriptor,
+                                   FSCRYPT_KEY_DESCRIPTOR_SIZE) == 0 &&
+                            refcount_inc_not_zero(&mk->mk_struct_refs))
+                                goto out;
+                }
+                break;
+        case FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER:
+                hlist_for_each_entry_rcu(mk, bucket, mk_node) {
+                        if (mk->mk_spec.type ==
+                                FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER &&
+                            memcmp(mk->mk_spec.u.identifier,
+                                   mk_spec->u.identifier,
+                                   FSCRYPT_KEY_IDENTIFIER_SIZE) == 0 &&
+                            refcount_inc_not_zero(&mk->mk_struct_refs))
+                                goto out;
+                }
+                break;
+        }
+        mk = NULL;
+out:
+        rcu_read_unlock();
+        return mk;
 }

 static int allocate_master_key_users_keyring(struct fscrypt_master_key *mk)
@@ -277,17 +338,30 @@ static int allocate_master_key_users_keyring(struct fscrypt_master_key *mk)
 static struct key *find_master_key_user(struct fscrypt_master_key *mk)
 {
        char description[FSCRYPT_MK_USER_DESCRIPTION_SIZE];
+       key_ref_t keyref;

        format_mk_user_description(description, mk->mk_spec.u.identifier);
-       return search_fscrypt_keyring(mk->mk_users, &key_type_fscrypt_user,
-                                     description);
+
+       /*
+        * We need to mark the keyring reference as "possessed" so that we
+        * acquire permission to search it, via the KEY_POS_SEARCH permission.
+        */
+       keyref = keyring_search(make_key_ref(mk->mk_users, true /*possessed*/),
+                               &key_type_fscrypt_user, description, false);
+       if (IS_ERR(keyref)) {
+               if (PTR_ERR(keyref) == -EAGAIN || /* not found */
+                   PTR_ERR(keyref) == -EKEYREVOKED) /* recently invalidated */
+                       keyref = ERR_PTR(-ENOKEY);
+               return ERR_CAST(keyref);
+       }
+       return key_ref_to_ptr(keyref);
 }

 /*
  * Give the current user a "key" in ->mk_users.  This charges the user's quota
  * and marks the master key as added by the current user, so that it cannot be
- * removed by another user with the key.  Either the master key's key->sem must
- * be held for write, or the master key must be still undergoing initialization.
```

```
+ * removed by another user with the key.  Either ->mk_sem must be held for
+ * write, or the master key must be still undergoing initialization.
  */
 static int add_master_key_user(struct fscrypt_master_key *mk)
 {
@@ -309,7 +383,7 @@ static int add_master_key_user(struct fscrypt_master_key *mk)

 /*
  * Remove the current user's "key" from ->mk_users.
- * The master key's key->sem must be held for write.
+ * ->mk_sem must be held for write.
  *
  * Returns 0 if removed, -ENOKEY if not found, or another -errno code.
  */
@@ -327,63 +401,49 @@ static int remove_master_key_user(struct fscrypt_master_key *mk)
 }

 /*
- * Allocate a new fscrypt_master_key which contains the given secret, set it as
- * the payload of a new 'struct key' of type fscrypt, and link the 'struct key'
- * into the given keyring.  Synchronized by fscrypt_add_key_mutex.
+ * Allocate a new fscrypt_master_key, transfer the given secret over to it, and
+ * insert it into sb->s_master_keys.
  */
-static int add_new_master_key(struct fscrypt_master_key_secret *secret,
-                              const struct fscrypt_key_specifier *mk_spec,
-                              struct key *keyring)
+static int add_new_master_key(struct super_block *sb,
+                              struct fscrypt_master_key_secret *secret,
+                              const struct fscrypt_key_specifier *mk_spec)
 {
+       struct fscrypt_keyring *keyring = sb->s_master_keys;
        struct fscrypt_master_key *mk;
-       char description[FSCRYPT_MK_DESCRIPTION_SIZE];
-       struct key *key;
        int err;

        mk = kzalloc(sizeof(*mk), GFP_KERNEL);
        if (!mk)
                return -ENOMEM;

+       mk->mk_sb = sb;
+       init_rwsem(&mk->mk_sem);
+       refcount_set(&mk->mk_struct_refs, 1);
        mk->mk_spec = *mk_spec;

-       move_master_key_secret(&mk->mk_secret, secret);
-
-       refcount_set(&mk->mk_refcount, 1); /* secret is present */
        INIT_LIST_HEAD(&mk->mk_decrypted_inodes);
        spin_lock_init(&mk->mk_decrypted_inodes_lock);

        if (mk_spec->type == FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER) {
                err = allocate_master_key_users_keyring(mk);
                if (err)
-                       goto out_free_mk;
+                       goto out_put;
                err = add_master_key_user(mk);
                if (err)
-                       goto out_free_mk;
+                       goto out_put;
        }

-       /*
-        * Note that we don't charge this key to anyone's quota, since when
```

```
-		 * ->mk_users is in use those keys are charged instead, and otherwise
-		 * (when ->mk_users isn't in use) only root can add these keys.
-		 */
-		format_mk_description(description, mk_spec);
-		key = key_alloc(&key_type_fscrypt, description,
-				GLOBAL_ROOT_UID, GLOBAL_ROOT_GID, current_cred(),
-				KEY_POS_SEARCH | KEY_USR_SEARCH | KEY_USR_VIEW,
-				KEY_ALLOC_NOT_IN_QUOTA, NULL);
-		if (IS_ERR(key)) {
-			err = PTR_ERR(key);
-			goto out_free_mk;
-		}
-		err = key_instantiate_and_link(key, mk, sizeof(*mk), keyring, NULL);
-		key_put(key);
-		if (err)
-			goto out_free_mk;
+		move_master_key_secret(&mk->mk_secret, secret);
+		refcount_set(&mk->mk_active_refs, 1); /* ->mk_secret is present */

+		spin_lock(&keyring->lock);
+		hlist_add_head_rcu(&mk->mk_node,
+				   fscrypt_mk_hash_bucket(keyring, mk_spec));
+		spin_unlock(&keyring->lock);
 		return 0;

-out_free_mk:
-		free_master_key(mk);
+out_put:
+		fscrypt_put_master_key(mk);
 		return err;
 }

@@ -392,42 +452,34 @@ out_free_mk:
 static int add_existing_master_key(struct fscrypt_master_key *mk,
 				   struct fscrypt_master_key_secret *secret)
 {
-		struct key *mk_user;
-		bool rekey;
 		int err;

 		/*
 		 * If the current user is already in ->mk_users, then there's nothing to
-		 * do.  (Not applicable for v1 policy keys, which have NULL ->mk_users.)
+		 * do.  Otherwise, we need to add the user to ->mk_users.  (Neither is
+		 * applicable for v1 policy keys, which have NULL ->mk_users.)
 		 */
 		if (mk->mk_users) {
-			mk_user = find_master_key_user(mk);
+			struct key *mk_user = find_master_key_user(mk);
+
 			if (mk_user != ERR_PTR(-ENOKEY)) {
 				if (IS_ERR(mk_user))
 					return PTR_ERR(mk_user);
 				key_put(mk_user);
 				return 0;
 			}
-		}
-
-		/* If we'll be re-adding ->mk_secret, try to take the reference. */
-		rekey = !is_master_key_secret_present(&mk->mk_secret);
-		if (rekey && !refcount_inc_not_zero(&mk->mk_refcount))
-			return KEY_DEAD;
-
-		/* Add the current user to ->mk_users, if applicable. */
-		if (mk->mk_users) {
```

```
                err = add_master_key_user(mk);
-               if (err) {
-                       if (rekey && refcount_dec_and_test(&mk->mk_refcount))
-                               return KEY_DEAD;
+               if (err)
                        return err;
-               }
        }

        /* Re-add the secret if needed. */
-       if (rekey)
+       if (!is_master_key_secret_present(&mk->mk_secret)) {
+               if (!refcount_inc_not_zero(&mk->mk_active_refs))
+                       return KEY_DEAD;
                move_master_key_secret(&mk->mk_secret, secret);
+       }
+
        return 0;
 }

@@ -436,38 +488,36 @@ static int do_add_master_key(struct super_block *sb,
                             const struct fscrypt_key_specifier *mk_spec)
 {
        static DEFINE_MUTEX(fscrypt_add_key_mutex);
-       struct key *key;
+       struct fscrypt_master_key *mk;
        int err;

        mutex_lock(&fscrypt_add_key_mutex); /* serialize find + link */
-retry:
-       key = fscrypt_find_master_key(sb, mk_spec);
-       if (IS_ERR(key)) {
-               err = PTR_ERR(key);
-               if (err != -ENOKEY)
-                       goto out_unlock;
+
+       mk = fscrypt_find_master_key(sb, mk_spec);
+       if (!mk) {
                /* Didn't find the key in ->s_master_keys.  Add it. */
                err = allocate_filesystem_keyring(sb);
-               if (err)
-                       goto out_unlock;
-               err = add_new_master_key(secret, mk_spec, sb->s_master_keys);
+               if (!err)
+                       err = add_new_master_key(sb, secret, mk_spec);
        } else {
                /*
                 * Found the key in ->s_master_keys.  Re-add the secret if
                 * needed, and add the user to ->mk_users if needed.
                 */
-               down_write(&key->sem);
-               err = add_existing_master_key(key->payload.data[0], secret);
-               up_write(&key->sem);
+               down_write(&mk->mk_sem);
+               err = add_existing_master_key(mk, secret);
+               up_write(&mk->mk_sem);
                if (err == KEY_DEAD) {
-                       /* Key being removed or needs to be removed */
-                       key_invalidate(key);
-                       key_put(key);
-                       goto retry;
+                       /*
+                        * We found a key struct, but it's already been fully
+                        * removed.  Ignore the old struct and add a new one.
+                        * fscrypt_add_key_mutex means we don't need to worry
```

```
+                        * about concurrent adds.
+                        */
+                       err = add_new_master_key(sb, secret, mk_spec);
                }
-               key_put(key);
+               fscrypt_put_master_key(mk);
        }
-out_unlock:
        mutex_unlock(&fscrypt_add_key_mutex);
        return err;
 }
@@ -771,19 +821,19 @@ int fscrypt_verify_key_added(struct super_block *sb,
                             const u8 identifier[FSCRYPT_KEY_IDENTIFIER_SIZE])
 {
        struct fscrypt_key_specifier mk_spec;
-       struct key *key, *mk_user;
        struct fscrypt_master_key *mk;
+       struct key *mk_user;
        int err;

        mk_spec.type = FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER;
        memcpy(mk_spec.u.identifier, identifier, FSCRYPT_KEY_IDENTIFIER_SIZE);

-       key = fscrypt_find_master_key(sb, &mk_spec);
-       if (IS_ERR(key)) {
-               err = PTR_ERR(key);
+       mk = fscrypt_find_master_key(sb, &mk_spec);
+       if (!mk) {
+               err = -ENOKEY;
                goto out;
        }
-       mk = key->payload.data[0];
+       down_read(&mk->mk_sem);
        mk_user = find_master_key_user(mk);
        if (IS_ERR(mk_user)) {
                err = PTR_ERR(mk_user);
@@ -791,7 +841,8 @@ int fscrypt_verify_key_added(struct super_block *sb,
                key_put(mk_user);
                err = 0;
        }
-       key_put(key);
+       up_read(&mk->mk_sem);
+       fscrypt_put_master_key(mk);
 out:
        if (err == -ENOKEY && capable(CAP_FOWNER))
                err = 0;
@@ -953,11 +1004,10 @@ static int do_remove_key(struct file *filp, void __user *_uarg, bool all_users)
        struct super_block *sb = file_inode(filp)->i_sb;
        struct fscrypt_remove_key_arg __user *uarg = _uarg;
        struct fscrypt_remove_key_arg arg;
-       struct key *key;
        struct fscrypt_master_key *mk;
        u32 status_flags = 0;
        int err;
-       bool dead;
+       bool inodes_remain;

        if (copy_from_user(&arg, uarg, sizeof(arg)))
                return -EFAULT;
@@ -977,12 +1027,10 @@ static int do_remove_key(struct file *filp, void __user *_uarg, bool all_users)
                return -EACCES;

        /* Find the key being removed. */
-       key = fscrypt_find_master_key(sb, &arg.key_spec);
-       if (IS_ERR(key))
```

```
-                return PTR_ERR(key);
-        mk = key->payload.data[0];
-
-        down_write(&key->sem);
+        mk = fscrypt_find_master_key(sb, &arg.key_spec);
+        if (!mk)
+                return -ENOKEY;
+        down_write(&mk->mk_sem);

         /* If relevant, remove current user's (or all users) claim to the key */
         if (mk->mk_users && mk->mk_users->keys.nr_leaves_on_tree != 0) {
@@ -991,7 +1039,7 @@ static int do_remove_key(struct file *filp, void __user *_uarg, bool all_users)
                 else
                         err = remove_master_key_user(mk);
                 if (err) {
-                        up_write(&key->sem);
+                        up_write(&mk->mk_sem);
                         goto out_put_key;
                 }
                 if (mk->mk_users->keys.nr_leaves_on_tree != 0) {
@@ -1003,26 +1051,22 @@ static int do_remove_key(struct file *filp, void __user *_uarg, bool all_users)
                         status_flags |=
                                 FSCRYPT_KEY_REMOVAL_STATUS_FLAG_OTHER_USERS;
                         err = 0;
-                        up_write(&key->sem);
+                        up_write(&mk->mk_sem);
                         goto out_put_key;
                 }
         }

         /* No user claims remaining.  Go ahead and wipe the secret. */
-        dead = false;
+        err = -ENOKEY;
         if (is_master_key_secret_present(&mk->mk_secret)) {
                 wipe_master_key_secret(&mk->mk_secret);
-                dead = refcount_dec_and_test(&mk->mk_refcount);
-        }
-        up_write(&key->sem);
-        if (dead) {
-                /*
-                 * No inodes reference the key, and we wiped the secret, so the
-                 * key object is free to be removed from the keyring.
-                 */
-                key_invalidate(key);
+                fscrypt_put_master_key_activeref(mk);
                 err = 0;
-        } else {
+        }
+        inodes_remain = refcount_read(&mk->mk_active_refs) > 0;
+        up_write(&mk->mk_sem);
+
+        if (inodes_remain) {
                 /* Some inodes still reference this key; try to evict them. */
                 err = try_to_lock_encrypted_files(sb, mk);
                 if (err == -EBUSY) {
@@ -1038,7 +1082,7 @@ static int do_remove_key(struct file *filp, void __user *_uarg, bool all_users)
          * has been fully removed including all files locked.
          */
 out_put_key:
-        key_put(key);
+        fscrypt_put_master_key(mk);
         if (err == 0)
                 err = put_user(status_flags, &uarg->removal_status_flags);
         return err;
@@ -1085,7 +1129,6 @@ int fscrypt_ioctl_get_key_status(struct file *filp, void __user *uarg)
```

```
 {
        struct super_block *sb = file_inode(filp)->i_sb;
        struct fscrypt_get_key_status_arg arg;
-       struct key *key;
        struct fscrypt_master_key *mk;
        int err;

@@ -1102,19 +1145,18 @@ int fscrypt_ioctl_get_key_status(struct file *filp, void __user *uarg)
        arg.user_count = 0;
        memset(arg.__out_reserved, 0, sizeof(arg.__out_reserved));

-       key = fscrypt_find_master_key(sb, &arg.key_spec);
-       if (IS_ERR(key)) {
-               if (key != ERR_PTR(-ENOKEY))
-                       return PTR_ERR(key);
+       mk = fscrypt_find_master_key(sb, &arg.key_spec);
+       if (!mk) {
                arg.status = FSCRYPT_KEY_STATUS_ABSENT;
                err = 0;
                goto out;
        }
-       mk = key->payload.data[0];
-       down_read(&key->sem);
+       down_read(&mk->mk_sem);

        if (!is_master_key_secret_present(&mk->mk_secret)) {
-               arg.status = FSCRYPT_KEY_STATUS_INCOMPLETELY_REMOVED;
+               arg.status = refcount_read(&mk->mk_active_refs) > 0 ?
+                       FSCRYPT_KEY_STATUS_INCOMPLETELY_REMOVED :
+                       FSCRYPT_KEY_STATUS_ABSENT /* raced with full removal */;
                err = 0;
                goto out_release_key;
        }
@@ -1136,8 +1178,8 @@ int fscrypt_ioctl_get_key_status(struct file *filp, void __user *uarg)
        }
        err = 0;
 out_release_key:
-       up_read(&key->sem);
-       key_put(key);
+       up_read(&mk->mk_sem);
+       fscrypt_put_master_key(mk);
 out:
        if (!err && copy_to_user(uarg, &arg, sizeof(arg)))
                err = -EFAULT;
@@ -1149,13 +1191,9 @@ int __init fscrypt_init_keyring(void)
 {
        int err;

-       err = register_key_type(&key_type_fscrypt);
-       if (err)
-               return err;
-
        err = register_key_type(&key_type_fscrypt_user);
        if (err)
-               goto err_unregister_fscrypt;
+               return err;

        err = register_key_type(&key_type_fscrypt_provisioning);
        if (err)
@@ -1165,7 +1203,5 @@ int __init fscrypt_init_keyring(void)

 err_unregister_fscrypt_user:
        unregister_key_type(&key_type_fscrypt_user);
-err_unregister_fscrypt:
-       unregister_key_type(&key_type_fscrypt);
```

```
        return err;
 }

diff --git a/fs/crypto/keysetup.c b/fs/crypto/keysetup.c
index fbc71abdabe324..e037a7b8e9e42b 100644
--- a/fs/crypto/keysetup.c
+++ b/fs/crypto/keysetup.c
@@ -9,7 +9,6 @@
  */

 #include <crypto/skcipher.h>
-#include <linux/key.h>
 #include <linux/random.h>

 #include "fscrypt_private.h"
@@ -159,6 +158,7 @@ void fscrypt_destroy_prepared_key(struct fscrypt_prepared_key *prep_key)
 {
        crypto_free_skcipher(prep_key->tfm);
        fscrypt_destroy_inline_crypt_key(prep_key);
+       memzero_explicit(prep_key, sizeof(*prep_key));
 }

 /* Given a per-file encryption key, set up the file's crypto transform object */
@@ -412,20 +412,18 @@ static bool fscrypt_valid_master_key_size(const struct fscrypt_master_key *mk,
 /*
  * Find the master key, then set up the inode's actual encryption key.
  *
- * If the master key is found in the filesystem-level keyring, then the
- * corresponding 'struct key' is returned in *master_key_ret with its semaphore
- * read-locked.  This is needed to ensure that only one task links the
- * fscrypt_info into ->mk_decrypted_inodes (as multiple tasks may race to create
- * an fscrypt_info for the same inode), and to synchronize the master key being
- * removed with a new inode starting to use it.
+ * If the master key is found in the filesystem-level keyring, then it is
+ * returned in *mk_ret with its semaphore read-locked.  This is needed to ensure
+ * that only one task links the fscrypt_info into ->mk_decrypted_inodes (as
+ * multiple tasks may race to create an fscrypt_info for the same inode), and to
+ * synchronize the master key being removed with a new inode starting to use it.
  */
 static int setup_file_encryption_key(struct fscrypt_info *ci,
                                      bool need_dirhash_key,
-                                     struct key **master_key_ret)
+                                     struct fscrypt_master_key **mk_ret)
 {
-       struct key *key;
-       struct fscrypt_master_key *mk = NULL;
        struct fscrypt_key_specifier mk_spec;
+       struct fscrypt_master_key *mk;
        int err;

        err = fscrypt_select_encryption_impl(ci);
@@ -436,11 +434,10 @@ static int setup_file_encryption_key(struct fscrypt_info *ci,
        if (err)
                return err;

-       key = fscrypt_find_master_key(ci->ci_inode->i_sb, &mk_spec);
-       if (IS_ERR(key)) {
-               if (key != ERR_PTR(-ENOKEY) ||
-                   ci->ci_policy.version != FSCRYPT_POLICY_V1)
-                       return PTR_ERR(key);
+       mk = fscrypt_find_master_key(ci->ci_inode->i_sb, &mk_spec);
+       if (!mk) {
+               if (ci->ci_policy.version != FSCRYPT_POLICY_V1)
+                       return -ENOKEY;

                /*
```

```
                 * As a legacy fallback for v1 policies, search for the key in
@@ -450,9 +447,7 @@ static int setup_file_encryption_key(struct fscrypt_info *ci,
                 */
                return fscrypt_setup_v1_file_key_via_subscribed_keyrings(ci);
        }
-
-       mk = key->payload.data[0];
-       down_read(&key->sem);
+       down_read(&mk->mk_sem);

        /* Has the secret been removed (via FS_IOC_REMOVE_ENCRYPTION_KEY)? */
        if (!is_master_key_secret_present(&mk->mk_secret)) {
@@ -480,18 +475,18 @@ static int setup_file_encryption_key(struct fscrypt_info *ci,
        if (err)
                goto out_release_key;

-       *master_key_ret = key;
+       *mk_ret = mk;
        return 0;

 out_release_key:
-       up_read(&key->sem);
-       key_put(key);
+       up_read(&mk->mk_sem);
+       fscrypt_put_master_key(mk);
        return err;
 }

 static void put_crypt_info(struct fscrypt_info *ci)
 {
-       struct key *key;
+       struct fscrypt_master_key *mk;

        if (!ci)
                return;
@@ -501,24 +496,18 @@ static void put_crypt_info(struct fscrypt_info *ci)
        else if (ci->ci_owns_key)
                fscrypt_destroy_prepared_key(&ci->ci_enc_key);

-       key = ci->ci_master_key;
-       if (key) {
-               struct fscrypt_master_key *mk = key->payload.data[0];
-
+       mk = ci->ci_master_key;
+       if (mk) {
                /*
                 * Remove this inode from the list of inodes that were unlocked
-                * with the master key.
-                *
-                * In addition, if we're removing the last inode from a key that
-                * already had its secret removed, invalidate the key so that it
-                * gets removed from ->s_master_keys.
+                * with the master key.  In addition, if we're removing the last
+                * inode from a master key struct that already had its secret
+                * removed, then complete the full removal of the struct.
                 */
                spin_lock(&mk->mk_decrypted_inodes_lock);
                list_del(&ci->ci_master_key_link);
                spin_unlock(&mk->mk_decrypted_inodes_lock);
-               if (refcount_dec_and_test(&mk->mk_refcount))
-                       key_invalidate(key);
-               key_put(key);
+               fscrypt_put_master_key_activeref(mk);
        }
        memzero_explicit(ci, sizeof(*ci));
```

```
        kmem_cache_free(fscrypt_info_cachep, ci);
@@ -532,7 +521,7 @@ fscrypt_setup_encryption_info(struct inode *inode,
 {
        struct fscrypt_info *crypt_info;
        struct fscrypt_mode *mode;
-       struct key *master_key = NULL;
+       struct fscrypt_master_key *mk = NULL;
        int res;

        res = fscrypt_initialize(inode->i_sb->s_cop->flags);
@@ -555,8 +544,7 @@ fscrypt_setup_encryption_info(struct inode *inode,
        WARN_ON(mode->ivsize > FSCRYPT_MAX_IV_SIZE);
        crypt_info->ci_mode = mode;

-       res = setup_file_encryption_key(crypt_info, need_dirhash_key,
-                                       &master_key);
+       res = setup_file_encryption_key(crypt_info, need_dirhash_key, &mk);
        if (res)
                goto out;

@@ -571,12 +559,9 @@ fscrypt_setup_encryption_info(struct inode *inode,
                 * We won the race and set ->i_crypt_info to our crypt_info.
                 * Now link it into the master key's inode list.
                 */
-               if (master_key) {
-                       struct fscrypt_master_key *mk =
-                               master_key->payload.data[0];
-
-                       refcount_inc(&mk->mk_refcount);
-                       crypt_info->ci_master_key = key_get(master_key);
+               if (mk) {
+                       crypt_info->ci_master_key = mk;
+                       refcount_inc(&mk->mk_active_refs);
                        spin_lock(&mk->mk_decrypted_inodes_lock);
                        list_add(&crypt_info->ci_master_key_link,
                                 &mk->mk_decrypted_inodes);
@@ -586,9 +571,9 @@ fscrypt_setup_encryption_info(struct inode *inode,
        }
        res = 0;
 out:
-       if (master_key) {
-               up_read(&master_key->sem);
-               key_put(master_key);
+       if (mk) {
+               up_read(&mk->mk_sem);
+               fscrypt_put_master_key(mk);
        }
        put_crypt_info(crypt_info);
        return res;
@@ -753,7 +738,6 @@ EXPORT_SYMBOL(fscrypt_free_inode);
 int fscrypt_drop_inode(struct inode *inode)
 {
        const struct fscrypt_info *ci = fscrypt_get_info(inode);
-       const struct fscrypt_master_key *mk;

        /*
         * If ci is NULL, then the inode doesn't have an encryption key set up
@@ -763,7 +747,6 @@ int fscrypt_drop_inode(struct inode *inode)
         */
        if (!ci || !ci->ci_master_key)
                return 0;
-       mk = ci->ci_master_key->payload.data[0];

        /*
         * With proper, non-racy use of FS_IOC_REMOVE_ENCRYPTION_KEY, all inodes
```

```
@@ -782,6 +765,6 @@ int fscrypt_drop_inode(struct inode *inode)
         * then the thread removing the key will either evict the inode itself
         * or will correctly detect that it wasn't evicted due to the race.
         */
-       return !is_master_key_secret_present(&mk->mk_secret);
+       return !is_master_key_secret_present(&ci->ci_master_key->mk_secret);
 }
 EXPORT_SYMBOL_GPL(fscrypt_drop_inode);
```

```
diff --git a/fs/crypto/policy.c b/fs/crypto/policy.c
index 80b8ca0f340b29..8485e7eaee2b3d 100644
--- a/fs/crypto/policy.c
+++ b/fs/crypto/policy.c
@@ -744,12 +744,8 @@ int fscrypt_set_context(struct inode *inode, void *fs_data)
         * delayed key setup that requires the inode number.
         */
        if (ci->ci_policy.version == FSCRYPT_POLICY_V2 &&
-           (ci->ci_policy.v2.flags & FSCRYPT_POLICY_FLAG_IV_INO_LBLK_32)) {
-               const struct fscrypt_master_key *mk =
-                       ci->ci_master_key->payload.data[0];
-
-               fscrypt_hash_inode_number(ci, mk);
-       }
+           (ci->ci_policy.v2.flags & FSCRYPT_POLICY_FLAG_IV_INO_LBLK_32))
+               fscrypt_hash_inode_number(ci, ci->ci_master_key);

        return inode->i_sb->s_cop->set_context(inode, &ctx, ctxsize, fs_data);
 }
```

```
diff --git a/fs/super.c b/fs/super.c
index 734ed584a946e1..6a82660e1adba8 100644
--- a/fs/super.c
+++ b/fs/super.c
@@ -291,7 +291,6 @@ static void __put_super(struct super_block *s)
                WARN_ON(s->s_inode_lru.node);
                WARN_ON(!list_empty(&s->s_mounts));
                security_sb_free(s);
-               fscrypt_sb_free(s);
                put_user_ns(s->s_user_ns);
                kfree(s->s_subtype);
                call_rcu(&s->rcu, destroy_super_rcu);
@@ -480,6 +479,7 @@ void generic_shutdown_super(struct super_block *sb)
                evict_inodes(sb);
                /* only nonzero refcount inodes can have marks */
                fsnotify_sb_delete(sb);
+               fscrypt_sb_delete(sb);
                security_sb_delete(sb);

                if (sb->s_dio_done_wq) {
```

```
diff --git a/include/linux/fs.h b/include/linux/fs.h
index 56a4b4b02477db..7203f5582fd449 100644
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -1472,7 +1472,7 @@ struct super_block {
        const struct xattr_handler **s_xattr;
 #ifdef CONFIG_FS_ENCRYPTION
        const struct fscrypt_operations *s_cop;
-       struct key              *s_master_keys; /* master crypto keys in use */
+       struct fscrypt_keyring  *s_master_keys; /* master crypto keys in use */
 #endif
 #ifdef CONFIG_FS_VERITY
        const struct fsverity_operations *s_vop;
```

```
diff --git a/include/linux/fscrypt.h b/include/linux/fscrypt.h
index 7d2f1e0f23b1fe..d86f43bd955029 100644
```

```
--- a/include/linux/fscrypt.h
+++ b/include/linux/fscrypt.h
@@ -312,7 +312,7 @@ fscrypt_free_dummy_policy(struct fscrypt_dummy_policy *dummy_policy)
 }

 /* keyring.c */
-void fscrypt_sb_free(struct super_block *sb);
+void fscrypt_sb_delete(struct super_block *sb);
 int fscrypt_ioctl_add_key(struct file *filp, void __user *arg);
 int fscrypt_add_test_dummy_key(struct super_block *sb,
                                const struct fscrypt_dummy_policy *dummy_policy);
@@ -526,7 +526,7 @@ fscrypt_free_dummy_policy(struct fscrypt_dummy_policy *dummy_policy)
 }

 /* keyring.c */
-static inline void fscrypt_sb_free(struct super_block *sb)
+static inline void fscrypt_sb_delete(struct super_block *sb)
 {
 }
```