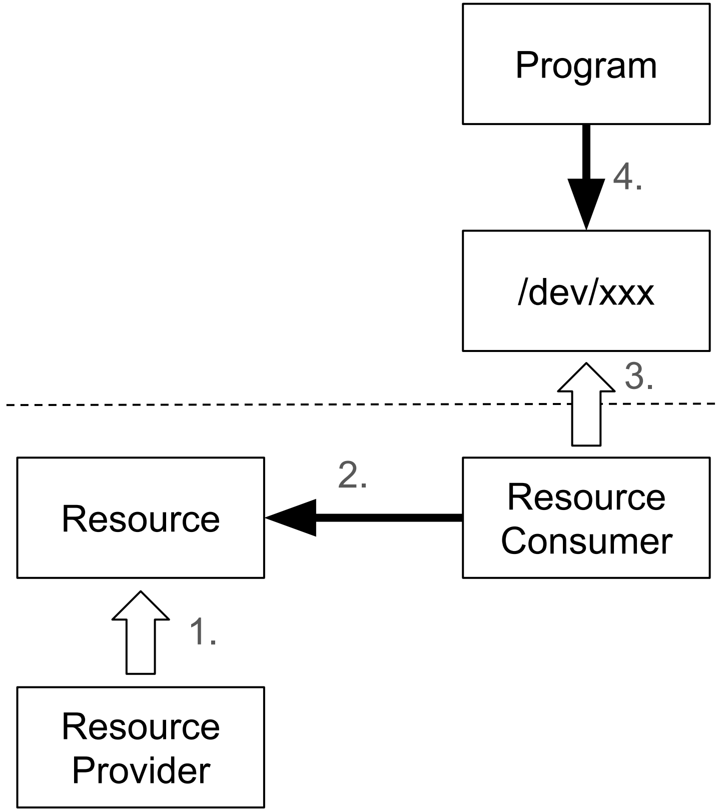
Revocable: a mechanism for preventing "classic" use-after-free bugs

Tzung-Bi Shih <tzungbi@kernel.org>

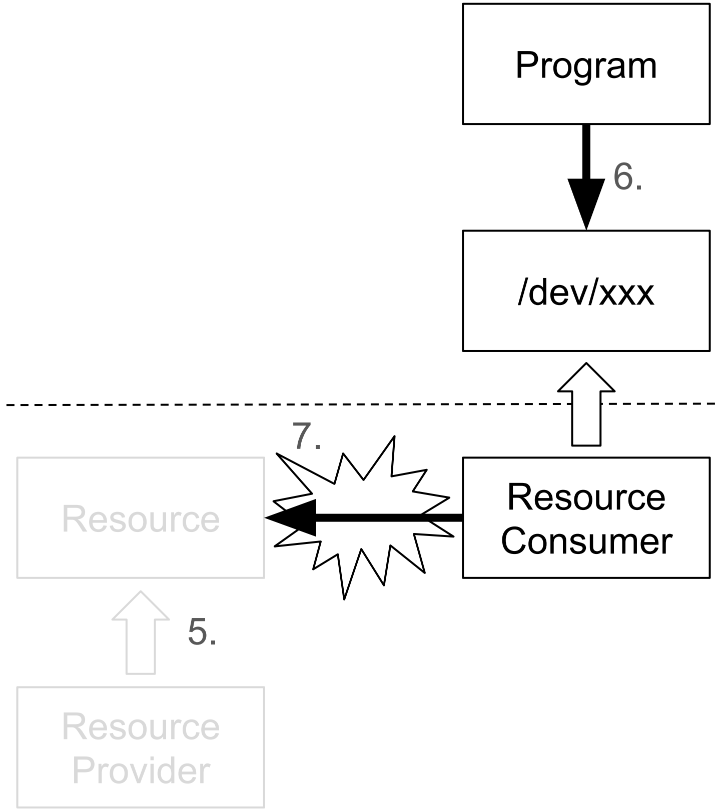
TOKYO, JAPAN / DECEMBER 11-13, 2025

The "classic" use-after-free (1/2)



1. A driver provides a resource.
2. Another driver relies on the resource.
3. The resource is exposed to userspace via an interface like a character device.
4. A userspace program opens the file and accesses the resource.

The "classic" use-after-free (2/2)

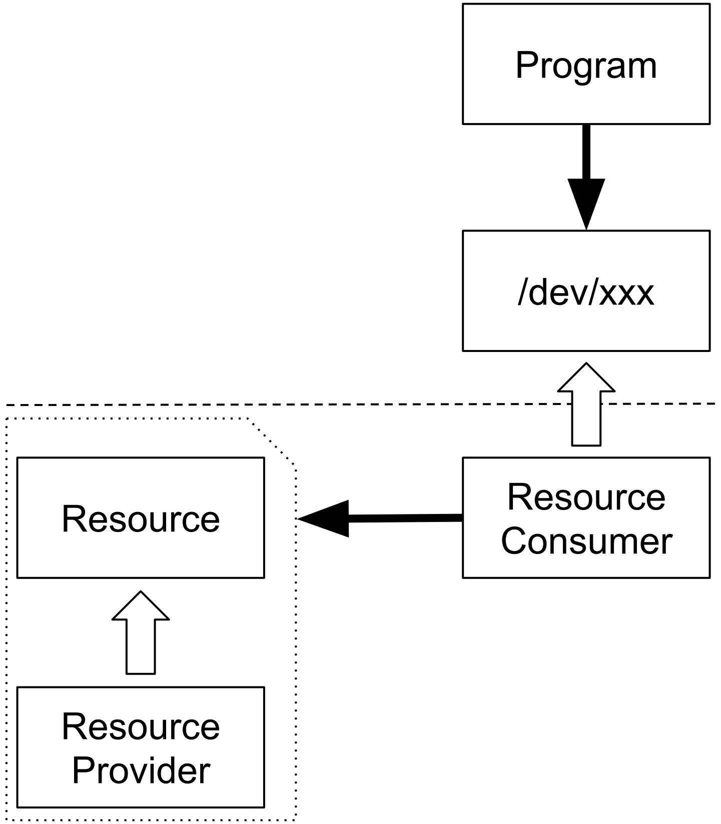


5. The provider driver is unbound and thus the resource is gone.
6. The userspace program accesses the resource.
7. The UAF happens.

Related works

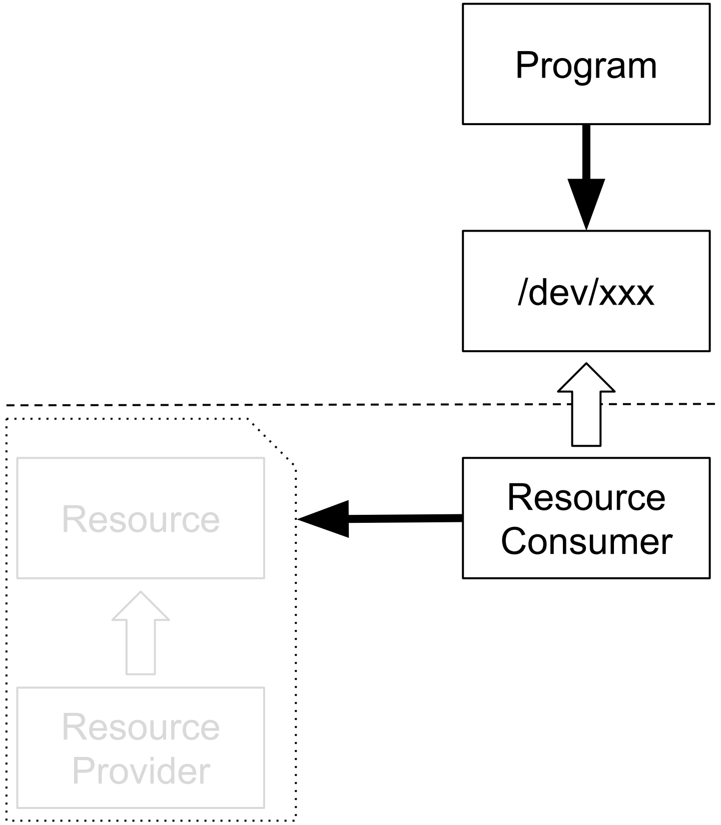
- Improving resource ownership and life-time in linux device drivers
  - Bartosz Golaszewski at LPC 2023
- Subsystems with object lifetime issues (in the embedded case)
  - Wolfram Sang at EOSS 2023
- Don't blame devres - devm_kzalloc() is not harmful
  - Bartosz Golaszewski at FOSDEM 2023
- Why is devm_kzalloc() harmful and what can we do about it
  - Laurent Pinchart at LPC 2022

Overview (1/2)



- Introduce an additional layer of indirection for resource access.
- Establish weak references.
- Decouple 2 independent lifecycles.

Overview (2/2)



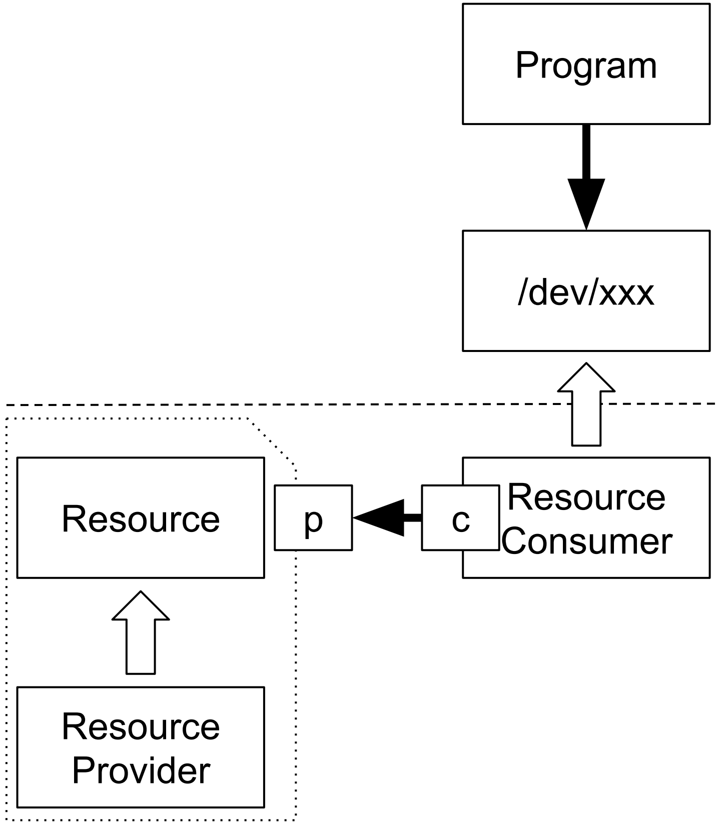
- Accessing the resource through the layer of indirection remains valid, even after the resource is gone.
- It returns NULL to signify that the resource is no longer available.

Devres vs. Revocable

	Devres	Revocable
Addressed issue	Resource leak	Invalid memory access
Focus on	Release resource	Invalidate access
When	Driver is unbound	The provider left

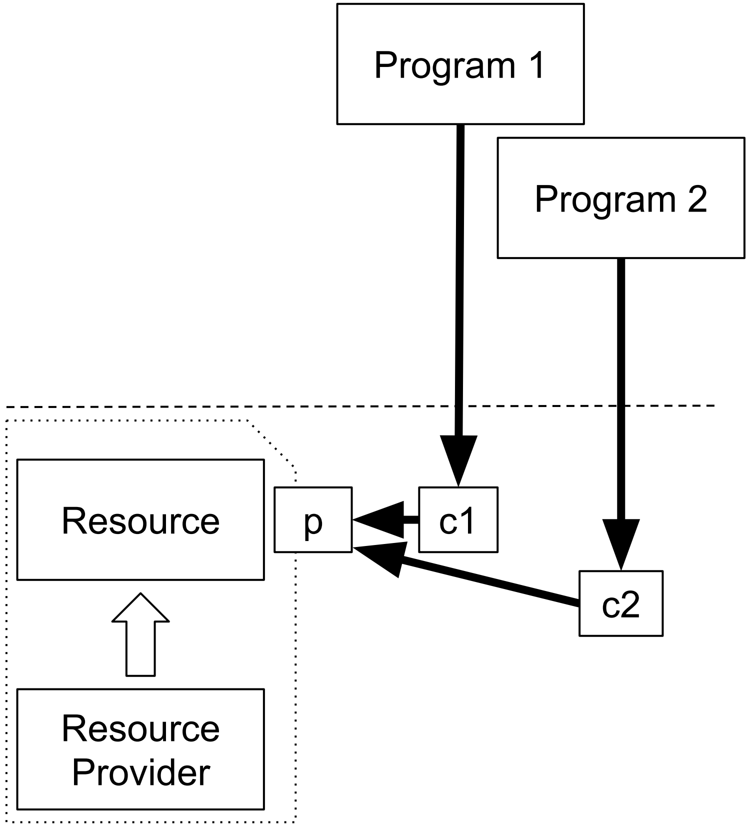
They are independent but can be used together.

Provider-consumer model



p	The handle of resource provider. A reference counted object. Persist as long as it still has references even if the resource has been revoked.
c	The handle of resource consumer. The same lifecycle with the consumer instance so that it is always available when the file is open.

Provider-consumer synchronization (1/2)

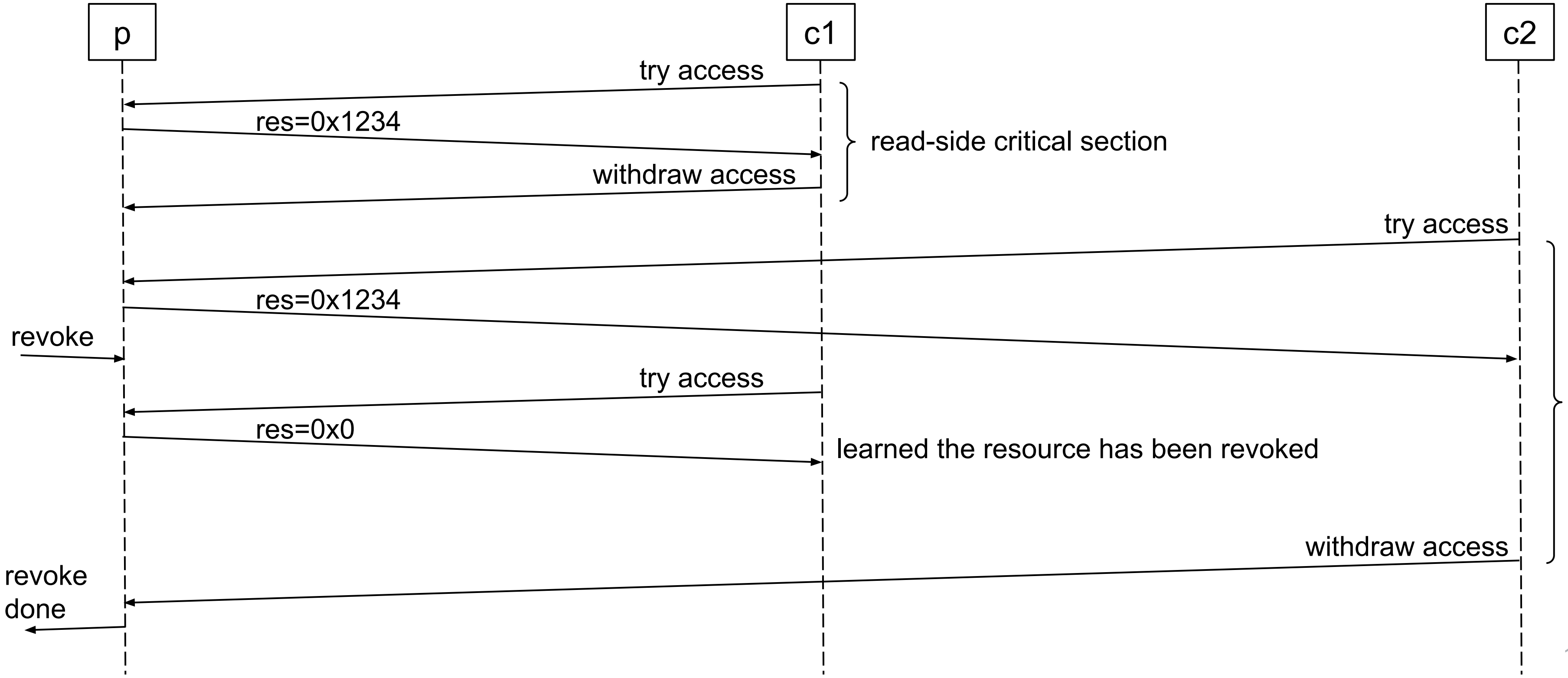


For the resource:

Writer	Reader
The resource provider	All instances of the resource consumer
1 instance	N instances
Infrequent, and only once when the resource is revoked	Frequent

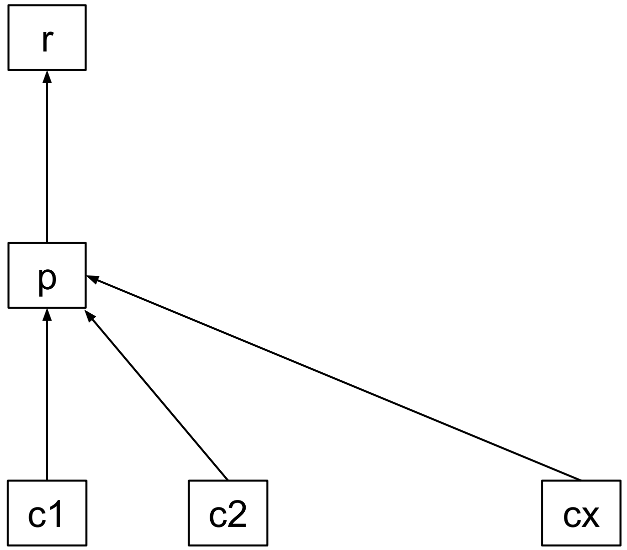
- RCU is the best fit.
- SRCU further extends the flexibility.

Provider-consumer synchronization (2/2)



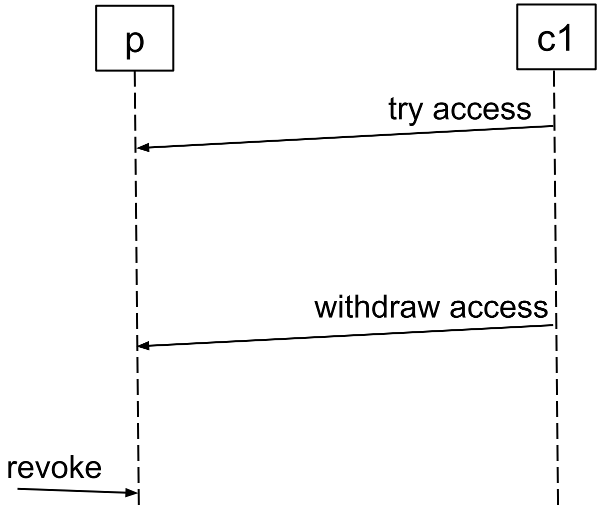
Implementation (1/2)

```
struct revocable_provider {
    struct srcu_struct srcu;
    void __rcu *res;
    struct kref kref;
};

struct revocable {
    struct revocable_provider *rp;
    int idx;
};

struct revocable *
revocable_alloc(struct revocable_provider *rp)
{
        ...

        kref_get(&rp->kref);
```


$$\begin{tikzcd} & r & & \\ & p \arrow[u] & & \\ c1 \arrow[ur] & c2 \arrow[u] & & cx \arrow[ull] \end{tikzcd}$$

Google

Implementation (2/2)



```
void *revocable_try_access(struct revocable *rev)
{
	struct revocable_provider *rp = rev->rp;

	rev->idx = srcu_read_lock(&rp->srcu);
	return srcu_dereference(rp->res, &rp->srcu);
}

void revocable_withdraw_access(struct revocable *rev)
{
	struct revocable_provider *rp = rev->rp;

	srcu_read_unlock(&rp->srcu, rev->idx);
}
```

```
void revocable_provider_revoke(struct revocable_provider *rp)
{
	rcu_assign_pointer(rp->res, NULL);
	synchronize_srcu(&rp->srcu);
	kref_put(&rp->kref, revocable_provider_release);
}
```

See v6 for more details.

Case study (1/2)

- We observed a trend of a class of UAF in ChromeOS.
    - Due to more EC-like devices come out
- It allocates a devm memory area when probing driver.

```
static int cros_ec_spi_probe(struct spi_device *spi)
{
...
        ec_spi = devm_kzalloc(dev, sizeof(*ec_spi), GFP_KERNEL);
        if (ec_spi == NULL)
                return -ENOMEM;
        ec_spi->spi = spi;
        ec_dev = cros_ec_device_alloc(dev);
```

```
struct cros_ec_device *cros_ec_device_alloc(struct device *dev)
{
        struct cros_ec_device *ec_dev;

        ec_dev = devm_kzalloc(dev, sizeof(*ec_dev), GFP_KERNEL);
```

- After the driver is unbound (e.g. due to the firmware crash), the memory (i.e. ec_dev) is of course freed.

Case study (2/2)

- However, there is still an opened file references the memory.

```
static const struct file_operations chardev_fops = {
        .open           = cros_ec_chardev_open,
        .poll           = cros_ec_chardev_poll,
        .read           = cros_ec_chardev_read,
        .release        = cros_ec_chardev_release,
```

```
static int cros_ec_chardev_open(struct inode *inode, struct file *filp)
{
        struct miscdevice *mdev = filp->private_data;
        struct cros_ec_dev *ec = dev_get_drvdata(mdev->parent);
        struct cros_ec_device *ec_dev = ec->ec_dev;
        struct chardev_priv *priv;
        int ret;

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

        priv->ec_dev = ec_dev;
```

```
static int cros_ec_chardev_release(struct inode *inode, struct file *filp)
{
        struct chardev_priv *priv = filp->private_data;
        struct cros_ec_device *ec_dev = priv->ec_dev;
        struct ec_event *event, *e;

        blocking_notifier_chain_unregister(&ec_dev->event_notifier,
                                           &priv->notifier);
```

Proposed solution 1: Use primitive APIs

```
@@ -166,7 +181,12 @@  static int cros_ec_chardev_open(struct inode
*inode, struct file *filp)
        if (!priv)
                return -ENOMEM;

-       priv->ec_dev = ec_dev;
+       priv->ec_dev_rev = revocable_alloc(ec_dev->revocable_provider);
```

```
@@ -64,7 +66,13 @@ static int ec_get_version(struct
chardev_priv *priv, char *str, int maxlen)
        msg->command = EC_CMD_GET_VERSION + priv->cmd_offset;
        msg->insize = sizeof(*resp);

-       ret = cros_ec_cmd_xfer_status(priv->ec_dev, msg);
+       REVOCABLE_TRY_ACCESS_WITH(priv->ec_dev_rev, ec_dev);
+       if (!ec_dev) {
+               ret = -ENODEV;
+               goto exit;
+       }
+
+       ret = cros_ec_cmd_xfer_status(ec_dev, msg);
```

```
@@ -299,10 +329,17 @@ static long cros_ec_chardev_ioctl_xcmd(struct
chardev_priv *priv, void __user *a
        }

        s_cmd->command += priv->cmd_offset;
-       ret = cros_ec_cmd_xfer(priv->ec_dev, s_cmd);
-       /* Only copy data to userland if data was received. */
-       if (ret < 0)
-               goto exit;
+       REVOCABLE_TRY_ACCESS_SCOPED(priv->ec_dev_rev, ec_dev) {
+               if (!ec_dev) {
+                       ret = -ENODEV;
+                       goto exit;
+               }
+
+               ret = cros_ec_cmd_xfer(ec_dev, s_cmd);
+               /* Only copy data to userland if data was received. */
+               if (ret < 0)
+                       goto exit;
+       }
```

See v6 for more details.

Proposed solution 2: Replace file operations

```
@@ -157,10 +159,16 @@  static int misc_open(struct inode *inode, struct file *file)
          */
         file->private_data = c;

-        err = 0;
         replace_fops(file, new_fops);
-        if (file->f_op->open)
+
+        if (file->f_op->open) {
                 err = file->f_op->open(inode, file);
+                if (err)
+                        goto fail;
+        }
+
+        if (c->revocable)
+                err = fs_revocable_replace(c->rp, file);
```

```
+int fs_revocable_replace(struct revocable_provider *rp, struct file *filp)
+{
...
+        fr->rev = revocable_alloc(rp);
+        if (!fr->rev)
+                goto free_fr;
+
...
+        if (fr->fops.read)
+                fr->fops.read = fs_revocable_read;
...
+        filp->f_op = &fr->fops;
```

```
+static ssize_t fs_revocable_read(struct file *filp, char __user *buffer,
+                                 size_t length, loff_t *offset)
+{
+        void *any;
+        CLASS(fops_replacement, fr)(filp);
+
+        REVOCABLE_TRY_ACCESS_WITH(fr->rev, any);
+        if (!any)
+                return -ENODEV;
+
+        return fr->orig_fops->read(filp, buffer, length, offset);
+}
```

On-going effort: probably see v5 for a comprehensive one; but v6 for the latest.

Proposed solution comparison (1/2)

Use primitive APIs	Replace file operations
The finest grain of accessing resource. The read-side critical sections are as small as possible.	Coarse grain. Preserve resource even if the file operations don't use the resource.
The user code is verbose.	Simple and ideally less error-prone.

Open issues with replacing file operations:

- Need to find an extra place to save the context which needs to be the same lifecycle with the struct file.
    - New field in struct file isn't promising as it looks performance sensitive.
    - i_cdev in struct inode isn't suitable and may not be universally available.
    - private_data in struct file isn't usable.

Proposed solution comparison (2/2)

- Don't take the proposed solutions as exclusive.
- Primitive API usage will be there anyway.
  - E.g. for some core kernel code.
- We're still on the way to figure out approaches for integrating the mechanism to subsystems.

Linux Plumbers Conference 2025

東京 2025 LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DECEMBER 11-13, 2025

東京 2025 LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DECEMBER 11-13, 2025

Linux Plumbers Conference 2025

東京 2025 LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DECEMBER 11-13, 2025

Thank you

Linux Plumbers Conference 2025

東京 2025 LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DECEMBER 11-13, 2025