Shadow Variables

Shadow variables are a simple way for livepatch modules to associate additional "shadow" data with existing data structures. Shadow data is allocated separately from parent data structures, which are left unmodified. The shadow variable API described in this document is used to allocate/add and remove/free shadow variables to/from their parents.

The implementation introduces a global, in-kernel hashtable that associates pointers to parent objects and a numeric identifier of the shadow data. The numeric identifier is a simple enumeration that may be used to describe shadow variable version, class or type, etc. More specifically, the parent pointer serves as the hashtable key while the numeric id subsequently filters hashtable queries. Multiple shadow variables may attach to the same parent object, but their numeric identifier distinguishes between them.

1. Brief API summary

(See the full API usage docbook notes in livepatch/shadow.c.)

A hashtable references all shadow variables. These references are stored and retrieved through a <obj, id> pair.

- The klp_shadow variable data structure encapsulates both tracking meta-data and shadow-data:
  - meta-data
    - obj - pointer to parent object
    - id - data identifier
  - data[] - storage for shadow data

It is important to note that the klp_shadow_alloc() and klp_shadow_get_or_alloc() are zeroing the variable by default. They also allow to call a custom constructor function when a non-zero value is needed. Callers should provide whatever mutual exclusion is required.

Note that the constructor is called under klp_shadow_lock spinlock. It allows to do actions that can be done only once when a new variable is allocated.

- klp_shadow_get() - retrieve a shadow variable data pointer - search hashtable for <obj, id> pair
- klp_shadow_alloc() - allocate and add a new shadow variable - search hashtable for <obj, id> pair
  - if exists
    - WARN and return NULL
  - if <obj, id> doesn't already exist
    - allocate a new shadow variable
    - initialize the variable using a custom constructor and data when provided
    - add <obj, id> to the global hashtable
- klp_shadow_get_or_alloc() - get existing or alloc a new shadow variable - search hashtable for <obj, id> pair
  - if exists
    - return existing shadow variable
  - if <obj, id> doesn't already exist
    - allocate a new shadow variable
    - initialize the variable using a custom constructor and data when provided
    - add <obj, id> pair to the global hashtable
- klp_shadow_free() - detach and free a <obj, id> shadow variable - find and remove a <obj, id> reference from global hashtable
  - if found
    - call destructor function if defined
    - free shadow variable
- klp_shadow_free_all() - detach and free all <_, id> shadow variables - find and remove any <_, id> references from global hashtable
  - if found
    - call destructor function if defined
    - free shadow variable

2. Use cases

(See the example shadow variable livepatch modules in samples/livepatch/ for full working demonstrations.)

For the following use-case examples, consider commit 1d147bfa6429 ("mac80211: fix AP powersave TX vs. wakeup race"), which added a spinlock to net/mac80211/sta_info.h :: struct sta_info. Each use-case example can be considered a stand-alone livepatch implementation of this fix.

Matching parent's lifecycle

If parent data structures are frequently created and destroyed, it may be easiest to align their shadow variables lifetimes to the same allocation and release functions. In this case, the parent data structure is typically allocated, initialized, then registered in some manner. Shadow variable allocation and setup can then be considered part of the parent's initialization and should be completed before the parent "goes live" (ie, any shadow variable get-API requests are made for this <obj, id> pair.)

For commit 1d147bfa6429, when a parent sta_info structure is allocated, allocate a shadow copy of the ps_lock pointer, then initialize it:

```
#define PS_LOCK 1
struct sta_info *sta_info_alloc(struct ieee80211_sub_if_data *sdata,
                                const u8 *addr, gfp_t gfp)
{
      struct sta_info *sta;
      spinlock_t *ps_lock;

      /* Parent structure is created */
      sta = kzalloc(sizeof(*sta) + hw->sta_data_size, gfp);

      /* Attach a corresponding shadow variable, then initialize it */
      ps_lock = klp_shadow_alloc(sta, PS_LOCK, sizeof(*ps_lock), gfp,
                                 NULL, NULL);
      if (!ps_lock)
              goto shadow_fail;
      spin_lock_init(ps_lock);
      ...
```

When requiring a ps_lock, query the shadow variable API to retrieve one for a specific struct sta_info::

```
void ieee80211_sta_ps_deliver_wakeup(struct sta_info *sta)
{
      spinlock_t *ps_lock;

      /* sync with ieee80211_tx_h_unicast_ps_buf */
      ps_lock = klp_shadow_get(sta, PS_LOCK);
      if (ps_lock)
              spin_lock(ps_lock);
      ...
```

When the parent sta_info structure is freed, first free the shadow variable:

```
void sta_info_free(struct ieee80211_local *local, struct sta_info *sta)
{
      klp_shadow_free(sta, PS_LOCK, NULL);
      kfree(sta);
      ...
```

In-flight parent objects

Sometimes it may not be convenient or possible to allocate shadow variables alongside their parent objects. Or a livepatch fix may require shadow variables for only a subset of parent object instances. In these cases, the klp_shadow_get_or_alloc() call can be used to attach shadow variables to parents already in-flight.

For commit 1d147bfa6429, a good spot to allocate a shadow spinlock is inside ieee80211_sta_ps_deliver_wakeup():

```
int ps_lock_shadow_ctor(void *obj, void *shadow_data, void *ctor_data)
{
      spinlock_t *lock = shadow_data;

      spin_lock_init(lock);
      return 0;
}

#define PS_LOCK 1
void ieee80211_sta_ps_deliver_wakeup(struct sta_info *sta)
{
      spinlock_t *ps_lock;

      /* sync with ieee80211_tx_h_unicast_ps_buf */
      ps_lock = klp_shadow_get_or_alloc(sta, PS_LOCK,
                      sizeof(*ps_lock), GFP_ATOMIC,
                      ps_lock_shadow_ctor, NULL);

      if (ps_lock)
              spin_lock(ps_lock);
      ...
```

This usage will create a shadow variable, only if needed, otherwise it will use one that was already created for this <obj, id> pair.

Like the previous use-case, the shadow spinlock needs to be cleaned up. A shadow variable can be freed just before its parent object is freed, or even when the shadow variable itself is no longer required.

Other use-cases

Shadow variables can also be used as a flag indicating that a data structure was allocated by new, livepatched code. In this case, it doesn't matter what data value the shadow variable holds, its existence suggests how to handle the parent object.

3. References

- https://github.com/dynup/kpatch

  The livepatch implementation is based on the kpatch version of shadow variables.

- http://files.mkgnu.net/files/dynamos/doc/papers/dynamos_eurosys_07.pdf

  Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels (Kritis Makris, Kyung Dong Ryu 2007) presented a datatype update technique called "shadow data structures".