Empowering confidential VMs in the cloud to use their own firmware upon instantiation

Vitaly Kuznetsov
Sr. Principal Software Engineer, Virtualization, Red Hat.
vkuznets@redhat.com.

Anirban (Ani) Sinha
Principal Software Engineer, Virtualization, Red Hat.
anisinha@redhat.com.

Alexander Graf
Principal Software Engineer, AWS.
graf@amazon.com.

KVM Forum 2024
September 22 2024
Brno, CZE.

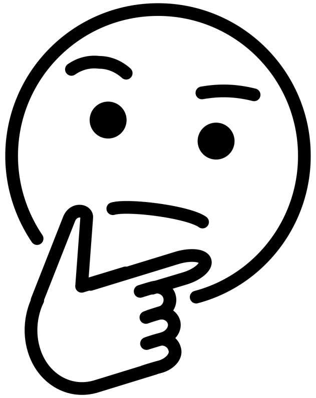
Red Hat

Focus of the talk

- What?
- Why?
- How?
- Demo
- Resources

Agenda

- **What?**
  - Why?
  - How?
  - Demo
  - Resources

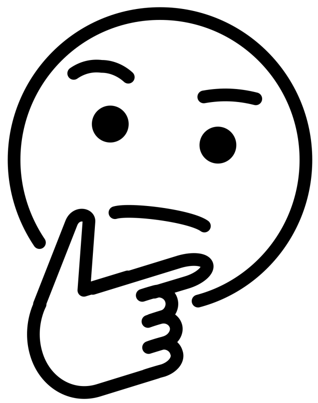


The talk is focused on in-guest firmware update mechanism

- The focus is on the situation when host administrator != guest tenant (e.g. ‘cloud’ use-case).
- The mechanism is mostly useful for Confidential VMs but in theory can work with traditional VMs too.

Red Hat

- What?
- **Why?**
- How?
- Demo
- Resources



Why guest tenants are interested in supplying their own firmware?

- Getting predictable, pre-calculated launch measurements
- Implementing exclusive per-guest features and configurations
  - Bring-your-own SecureBoot everything (and **trust** it!)
  - A stateful vTPM implementation with runtime attestations
  - ...
- and this is updateable during guest's lifecycle, not only upon creation.

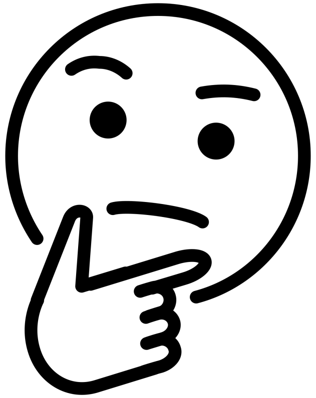
Why host owners are interested in providing the option to do in-guest firmware update?

- For Confidential VMs, updating firmware can’t go unnoticed by the guest tenant
  - Getting rid of the responsibility for non-trivial software which runs **inside** the confidential guest.
  - Guests may break because of new, unexpected launch measurements.
  - Tenants may be interested in what changed and in case of e.g. embargoed CVEs the information cannot be shared.

But why not just supply the firmware externally, as part of guest VM image or separately?

- The firmware will require a storage if supplied separately and this external storage will have to be linked to the VM's lifecycle.
- Storing the firmware as part of guest image (e.g. a file on ESP, separate partition,...) can be problematic:
  - The host may not have access to guest storage at all (e.g. NVME passthrough with acceleration card).

- What?
- Why?
- **How?**
- Demo
- Resources



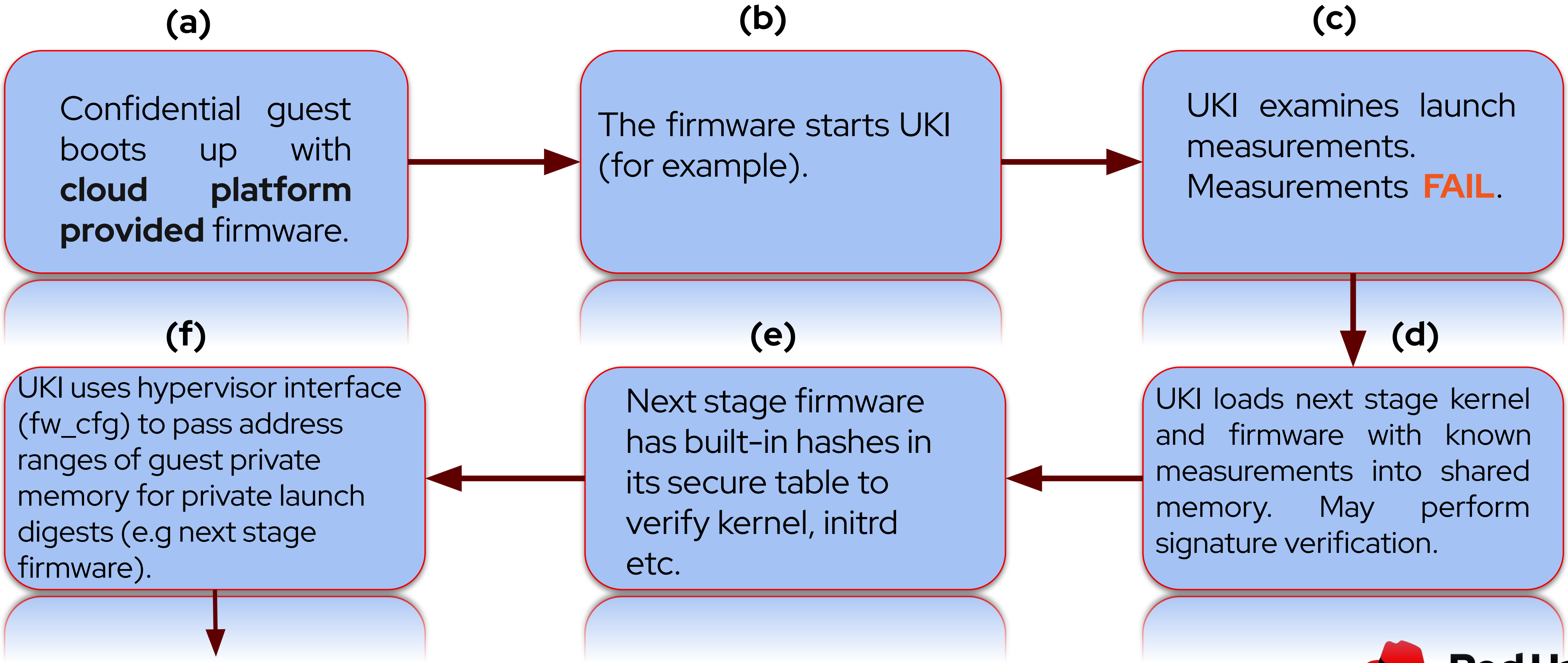
Brief overview of major stack components involved

- **QEMU**:
  - Hypervisor/guest interface in QEMU (fw_cfg based).
  - Guest reset mechanism for secure VMs
    - Currently in QEMU, resettings CPUS is not allowed - a reboot terminates the guest.
    - Shared memory needs to be preserved across reset.
    - New SEV context needs to be generated after reset.
  - Machine changes to make sure loader correctly loads firmware to the right address.

- **Systemd**:
  - Support for guest/hypervisor interface in systemd-boot.
  - Check platform/capabilities to make sure correct firmware is loaded.
  - Support for loading launch digests, using fw_cfg interface to pass digest information to hypervisor.
  - Trigger reset.

- **Firmware** (EDK2):
  - Fw_cfg changes to read platform/capability bits.
  - Scan fw_cfg vmfwupdate_blobs to find the kernel/initrd addresses and load linux from there.

Proposed launch digest update mechanism



Continued ...

Proposed launch digest update mechanism (contd ...)

Continued from previous slide ...

(g)

UKI passes the source address in shared memory where launch digests are loaded. Hypervisor copies private launch digests to guest private memory at address specified by the guest.

(h)

UKI may also pass target CPU state data for measurements to hypervisor using the same (fw_cfg) interface.

(i)

Hypervisor restarts the VM with the next stage launch digests (firmware, kernel etc) and target CPU state. Marks private memory as private. Generates new SEV context.

(j)

Firmware loads kernel, initrd etc with known measurements (since hashes for these components where built into the firmware).

(k)

After boot, PCR boot measurements are calculated and sent to external attestation server.

(l)

Attestation server verifies the measurements, they **PASS**. It provides the keys to unlock secrets.

Red Hat

Some further details …

Step (a):

- Stock firmware used by cloud provider is outside the trust zone of the end-users.
- Typically customers do not want any vendor provided component in the stack (between the guest and the hardware platform).

Step (c):

- Even if initial launch measurements with the vendor provided firmware is available, measurements will fail every time the vendor updates/changes the firmware (for example to put security fixes).
- This breaks customers.

Vitaly has covered it.

Step (d):

- UKI need to support "firmware" section (along with kernel, initrd, command line etc).

Some further details …

Step (e):

- Ukify.py ensures next stage firmware only loads trusted kernel and initrd etc by installing their hashes in the firmware's secure hash table.

Step (f):

- We are using the guest shared memory (shared with the hypervisor) as a data plane to pass the initial launch digests (firmware, kernel, initrd etc). The memory comes from the guest. No separate hypervisor memory allocation required.
- Private launch digests (eg. next stage firmware) are copied from the shared memory to guest private memory by the hypervisor before restarting and regenerating the VM context.
- Systemd checks platform/capability bits to make sure we are loading the correct firmware version for the correct platform.

Step (h):

- If we use default CPU reset register values, no need to pass initial CPU states.

Some further details ...

Step (j):

- Next stage firmware validates kernel, initrd etc since their hashes are stored in a hash table inside a secure page in the firmware.
  - There is no need to generate measurements for these components.
  - Signature verification also becomes optional.
- The firmware itself is validated by the launch measurements that are sent to the external attestation server.
- Firmware resides in the guest private memory pages. Rest can reside in the shared memory.

Step (l):

- To make sure that secure VM remains secure after updating the firmware, we also have a provision for implementing a "kill switch".
  - Once the firmware/kernel etc are updated, no more updates are allowed using the hypervisor interface.

Brief overview of major stack components involved (**revisit**)

- **QEMU**:
  - Hypervisor/guest interface in QEMU (fw_cfg based).
  - Guest reset mechanism for secure VMs
    - Currently in QEMU, resettings CPUS is not allowed - a reboot terminates the guest.
    - Shared memory needs to be preserved across reset.
    - New SEV context needs to be generated after reset.
  - Machine changes to make sure loader correctly loads firmware to the right address.

- **Systemd**:
  - Support for guest/hypervisor interface in systemd-boot.
  - Check platform/capabilities to make sure correct firmware is loaded.
  - Support for loading launch digests, using fw_cfg interface to pass digest information to hypervisor.
  - Trigger reset.

- **Firmware** (EDK2):
  - Fw_cfg changes to read platform/capability bits.
  - Scan fw_cfg vmfwupdate_blobs to find the kernel/initrd addresses and load linux from there.

QEMU hypervisor interface

```
typedef struct FwCfgVmFwUpdateBlob {
    /*
     * blob_type indicates the type of blob/launch digest the guest has passed
     * to the host. blob_type 0x00 is invalid. It is of type blob_type_t.
     */
    uint8_t blob_type;
    /*
     * map_type: type of guest memory mapping requested. Mappings can be either
     * private or shared. Private guest pages are flipped from shared to private
     * when a new SEV guest context is created. The private memory contains CPU
     * state information and firmware blob. The shared memory remains shared
     * with the hypervisor and is excluded from encryption and measurements.
     * The shared data is the next stage artifacts (kernel image/UKI, initrd,
     * command line) that are validated by the second stage firmware present in
     * the private memory. Thus they need not be explicitly measured by ASP.
     */
    uint8_t map_type;
    uint32_t size; /* size of the blob */
    uint64_t paddr; /* starting gpa where the blob is in guest memory. We
                     * copy the contents from the guest shared memory to a
                     * different guest private address target_paddr from paddr.
                     */
    uint64_t target_paddr; /* guest physical address where private blobs are
                            * copied to.
                            */
} FwCfgVmFwUpdateBlob;
```

QEMU hypervisor interface (contd ...)

```
/* type of mapping requested */
#define VMFW_TYPE_MAP_PRIVATE 0x00
#define VMFW_TYPE_MAP_SHARED 0x01

typedef enum {
    VMFW_TYPE_BLOB_KERNEL = 0x01, /* kernel */
    VMFW_TYPE_BLOB_INITRD, /* initrd */
    VMFW_TYPE_BLOB_CMDLINE, /* command line */
    VMFW_TYPE_BLOB_FW, /* firmware */
    VMFW_TYPE_BLOB_MAX
} blob_type_t;

typedef struct FwCfgVmFwUpdateCpuState {
    struct kvm_regs regs;
    /*
     * we are currently building this device only for x86.
     * So using sregs2 is fine even if its only available on x86.
     */
    struct kvm_sregs2 s;
} FwCfgVmFwUpdateCpuState;
```

QEMU hypervisor interface (contd ...)

```
struct VMFwUpdateState {
    DeviceState parent_obj;
    /*
     * platform and capabilities
     * least significant 3 bits - platform bits,
     * most significant 13 bits are capability bits.
     * Little endian format. Systemd loader checks these flags
     * before loading the firmware and kernel blobs to memory.
     */
    uint16_t platcap;
    /*
     * fw_cfg ctl
     *    - 't' - trigger vm regeneration.
     */
    uint8_t fw_cfg_ctl;

    /* number of blob entries passed by the guest */
    uint8_t n_entries;
    /*
     * Guest measurement blobs or launch digests - can be firmware blob,
     * kernel blob etc. Number of such blobs is stored in n_entries above.
     */
    FwCfgVmFwUpdateBlob vmfwupdate_blobs[MAX_VMFWUPD_ENTRIES];
    FwCfgVmFwUpdateCpuState cpu_state;
};
```

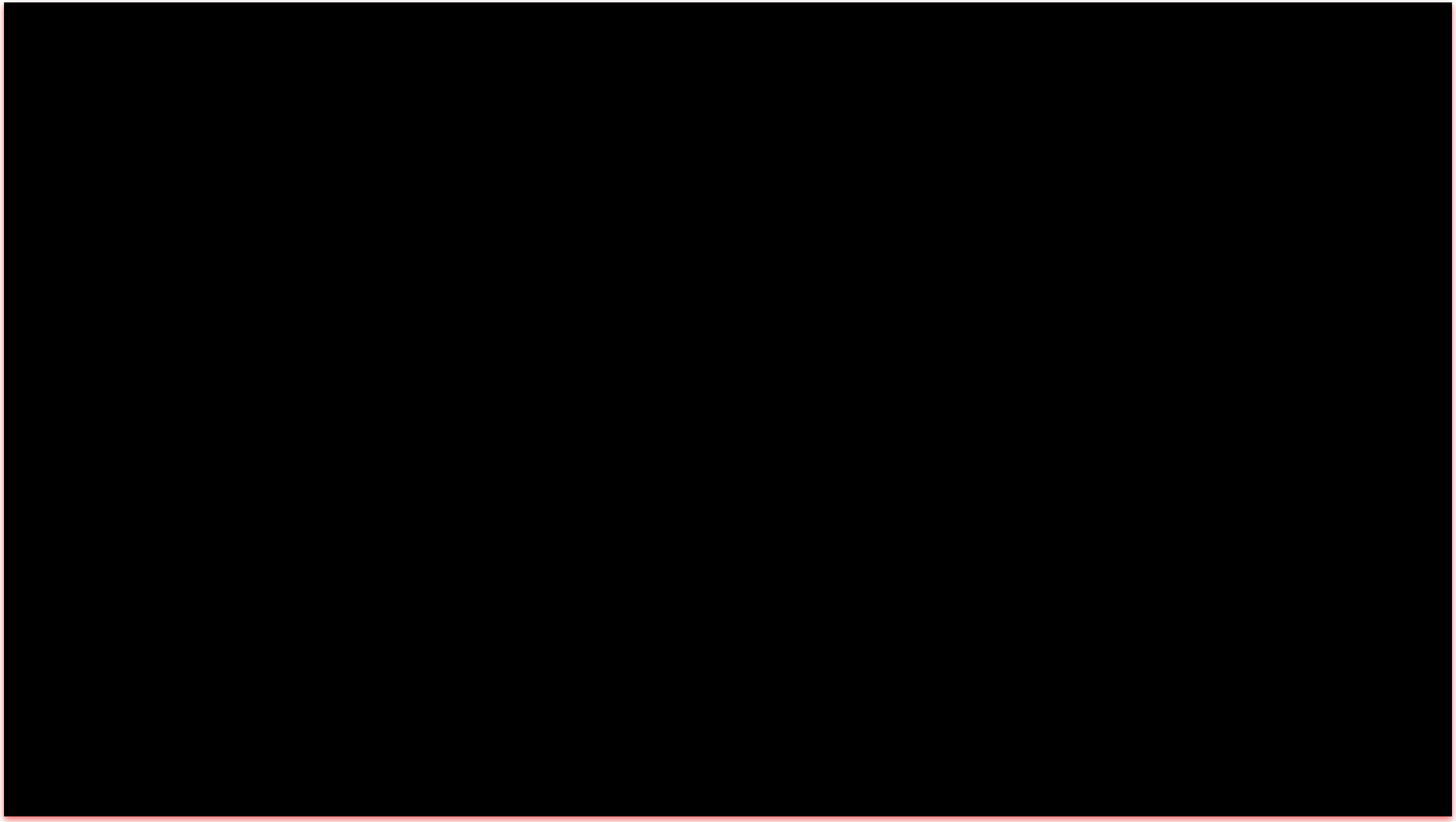
Systemd trigger using hypervisor interface

```
FIRMWARE_CONFIG_ITEM FwCfgItem;
size_t FwCfgSize;
if (QemuFwCfgFindFile("etc/vmfwupdate-blob", &FwCfgItem, &FwCfgSize) != EFI_SUCCESS)
{
        return EFI_LOAD_ERROR;
}
QemuFwCfgSelectItem(FwCfgItem);
QemuFwCfgWriteBytes(cur_blob * sizeof(FwCfgVmFwUpdateBlob), blobs);

if (QemuFwCfgFindFile("etc/fwupdate-control", &FwCfgItem, &FwCfgSize) != EFI_SUCCESS)
{
        return EFI_LOAD_ERROR;
}
QemuFwCfgSelectItem(FwCfgItem);
char cmd = 't';
QemuFwCfgWriteBytes(1, &cmd);

return EFI_LOAD_ERROR;
```

- What?
- Why?
- How?
- **Demo**
- Resources

Red Hat

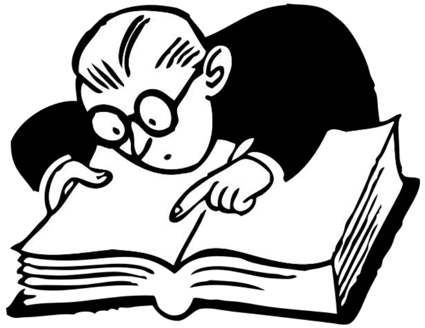
Demo of the concept in **non-CoCo** Virtual Machines



Getting the demo working ...

Getting the demo working ...

- What?
- Why?
- How?
- Demo
- **Resources**



Red Hat

Links to WIP/demo code

- QEMU changes:
- EDK changes:
- Systemd changes:
- Demo

Thanks to all who are involved!

- Thanks Vitaly Kuznetsov (Red Hat) for initiating this project within Red Hat, for guidance and for getting me (Ani Sinha) excited to jump in :-).
- Thanks Alex Graf (AWS) for the original idea, guidance and demo :-).
- Thanks Paolo Bonzini (Red Hat) for guidance and involvement in the project :-).
- Thanks Harald Hoyer (Matter Labs) for building FUKI support in systemd :-).

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat