Kernel Runtime Security Instrumentation

Florent Revest

FOSDEM 2020
Motivation
Our context

- Fleet of corporate Linux machines.
- Security monitoring and policies.
- Dynamic and scalable.
Example of signals

- A process that deletes its own executable.
- A Kernel module that loads and "hides" itself.
- "Suspicious" environment variables.
Example of mitigations

- Prevent known vulnerable binaries from running.
- Dynamic whitelist of known Kernel modules.
- Chain signals into a configurable MAC policy.
<table>
<thead>
<tr>
<th>Signals</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audit</td>
<td>SELinux, Apparmor</td>
</tr>
<tr>
<td>Perf</td>
<td>seccomp</td>
</tr>
</tbody>
</table>

- Audit: Logs system's behaviour
- Perf: Affects system's behaviour
Adding a new signal

<table>
<thead>
<tr>
<th>Signals</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audit</td>
<td>SELinux, Apparmor</td>
</tr>
<tr>
<td>Perf</td>
<td>seccomp</td>
</tr>
</tbody>
</table>

Update Audit (user/kernel) to log environment variables
Adding a new mitigation

**Signals**
- Audit
- Perf

**Mitigation**
- SELinux, Apparmor
- seccomp

Update the mitigation logic for a malicious actor with a known LD_PRELOAD signature
Introducing KRSI
LSM (Linux Security Module)

- **Basis of SELinux, AppArmor etc...**
  - Years of research and verification.

- Various hooks for system security **behaviours**.
  - Higher level than syscalls.

- Return value **allows or denies an operation**.
  - MAC (Mandatory Access Control)
BPF (Berkeley Packet Filter)

- **Bytecode** JIT-ed into the kernel.
- **Dynamically loaded** with libbpf.
- Can be **written in C**.
- **Statically verified** (e.g., read-only memory).
- Can **exchange data** with a userspace program.
BPF + LSM = KRSI

- Allows **BPF programs attachment to LSM hooks**.
- Dynamic **MAC and Audit policies** written in C.
- **PATCHv3 on LKML**
Usage example
Step 1: Hook into an appropriate LSM hook

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
    unsigned long reqprot, unsigned long prot)
{
    return 0;
}

return 0;
Step 2: Use eBPF helpers

```c
SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
            unsigned long reqprot, unsigned long prot)
{
    __u32 pid = bpf_get_current_pid_tgid();

    return 0;
}
```
Step 3: Access structure fields with BTF

```c
struct vm_area_struct {
    unsigned long vm_start;
} __attribute__((preserve_access_index));

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{
    __u32 pid = bpf_get_current_pid_tgid();
    unsigned long vm_start = vma->vm_start;

    return 0;
}
```
Step 4: Share variables with userspace

```c
int mprotect_count = 0;

struct vm_area_struct {
    unsigned long vm_start, vm_end;
} __attribute__((preserve_access_index));

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
        unsigned long reqprot, unsigned long prot)
{
    __u32 pid = bpf_get_current_pid_tgid();
    int vm_start = vma->vm_start;
    mprotect_count ++;
    return 0;
}
```
Step 5: Allow or deny an operation

```c
int mprotect_count = 0;

struct vm_area_struct {
    unsigned long vm_start, vm_end;
} __attribute__((preserve_access_index));

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{
    __u32 pid = bpf_get_current_pid_tgid();
    int vm_start = vma->vm_start;
    mprotect_count ++;
    return (mprotect_count > 100) ? -EPERM : 0;
}
```
What will you build next?

Thank You