NOVEMBER 4, 2025

REPORT 4 — CONCLUSION

FINAL RECOMMENDATIONS AND MITIGATION PLAN

NGUYỄN HOÀNG THANH PHONG – SE184915

Table of Contents

Executive summary

This project's triage and analysis chain (Report 1 — malware classification; Report 2 — dynamic analysis; Report 3 — static analysis) demonstrates that the analyzed artifacts are a loader-style Linux kernel module derived from Reptile-family code. The module (compiled test artifact reptile.ko) reconstructs / decrypts a payload at runtime, uses runtime symbol resolution (kallsyms-style), hooks syscall handlers and network hooks to hide artifacts, and spawns a hidden user-space shell (e.g., /reptile_shell) used for encrypted C2 and file exfiltration (e.g., /etc/passwd). Behavioral IOCs include calls to finit_module/insmod, hidden or altered syscall table entries, and network sessions (Attacker 192.168.1.130 ↔ Victim 192.168.1.150) including a spoofed source IP 192.168.0.2 and a listening port 4444. These results are documented in Reports 1–3.

This Report 4 maps those findings to the root causes (vulnerabilities exploited), prescribes prioritized preventive and detective controls (technical & operational), and provides a reproducible incident response / remediation playbook.

Scope & assumptions

Scope: Host-level mitigation and detection for Linux servers exposed to Reptile-like LKM rootkits; network controls for detecting C2 and exfiltration; incident response steps for suspected kernel compromise.

Assumptions: The victim host runs a distribution with dynamic module support (in analysis: CentOS7-like kernel `vermagic=3.10.0-1160.119.1`), and system administrators can change boot configuration, kernel cmdline, and install EVM/IMA/LKRG tools. Replace environment-specific values in the playbook below.

Threat model: attacker has privileged access (able to install kernel modules) or exploited a vulnerability enabling kernel code insertion; attacker can run arbitrary code, open encrypted channels, and perform IP spoofing within the target network.

Quick mapping: Finding -> Exploited vulnerabilities

Evidence	Mapped Vulnerability / Gap	Notes / Impact
`finit_module/insmod` observed in `strace`; copy of `/reptile/reptile.ko` staged	Kernel allows dynamic module loading	Demonstrates attacker can insert arbitrary kernel modules.
Kernel taint message: *"module verification failed: signature and/or required key missing"*	Unsigned modules accepted; module signing not enforced	Confirms permissive policy; no cryptographic enforcement.
Test module loaded under permissive debug kernel	Secure Boot/module verification not enforced	Shows bypass of intended protections.
Syscall table hooking; hidden `/reptile_shell` in memory; `/proc` listings inconsistent with Volatility	Lack of runtime kernel integrity monitoring	Kernel compromised at runtime; user-space tools (lsmod, ps) unreliable.
Encrypted / "TLS-like" transport on port 4444; IP spoofing to 192.168.0.2 during exfiltration	Network monitoring lacks deep TLS inspection / behavioral C2 detection	Malicious traffic evades simple ACLs and signature-based detection.

Prioritized countermeasures

I present recommended controls in three tiers: Preventive (stop installation), Detective (detect runtime compromise), and Remedial/Operational (respond & recover). For each control I give rationale, implementation steps (commands/config examples), and how to validate.

Top-level priority (must-do first): Enforce module signing + Secure Boot / kernel lockdown; deploy runtime kernel integrity (IMA/EVM and/or LKRG); implement process & memory forensics readiness.

1. Preventive controls

1.1. Enable and enforce kernel module signing (CONFIG_MODULE_SIG_FORCE)

Why: Prevents unsigned/out-of-tree modules (like original Reptile) from loading. Kernel will refuse to load modules without valid signatures.

How (high-level):

- Build/sign trusted module keys and enroll them in system UEFI MOK or kernel keyring. Example sign command from kernel sources: (see kernel docs)

```
scripts/sign-file sha512 <privkey> <pubcert> module.ko
```

- On systems with UEFI Secure Boot: enable Secure Boot and use mokutil to enroll your key if needed. Alternatively set `CONFIG_MODULE_SIG_FORCE=y` when building kernel to refuse unsigned modules.

Validation: Attempt to `insmod` an unsigned module and verify kernel rejects it (dmesg shows verification failure and reject). If kernel rejects unsigned modules, the sample could not be inserted.

1.2. Enable kernel lockdown / Secure Boot enforcement

Why: Kernel lockdown reduces attack surface and prevents certain operations even for root when enabled. When paired with module signing it prevents arbitrary drivers from loading.

How: Configure UEFI Secure Boot and ensure kernel is built with lockdown support or use distribution settings to enforce lockdown.

Validation: Validate via `cat /sys/kernel/security/lockdown`.

1.3. Consider disabling module loading if not required

Why: Systems that never need dynamic modules (appliances, hardened servers) should statically compile or disable module loading.

How: Runtime: `echo 1 > /proc/sys/kernel/modules_disabled` (reboot required for permanence); permanently configure kernel with `CONFIG_MODULES=n` for immutable environments.

Caveat: This impacts kernel upgrades and some drivers; apply only when operationally acceptable.

2. Runtime integrity & detection

2.1. Deploy IMA/EVM (Integrity Measurement Architecture + Extended Verification Module)

Why: IMA measures and (optionally) appraises files, preventing the execution of altered/untrusted binaries or modules; EVM protects critical extended attributes. This detects or prevents unauthorized module files and modified system binaries.

How: Add kernel cmdline: `ima_policy=tcb ima_appraise=fix evm=fix` and ensure keys loaded into kernel keyring; enroll keys in TPM or MOK; configure `ima-evm-utils` to set xattrs for trusted files. Use Red Hat docs for exact steps.

Validation: Attempt to load a modified signed file and verify appraisal failure is logged/audited.

2.2. Install a runtime kernel integrity monitor (LKRG)

Why: LKRG performs continuous integrity checks on kernel code areas and can detect or block changes such as syscall table hooking. It is effective at flagging in-memory tampering similar to Reptile's behavior.

How: Build the latest LKRG for your kernel and enable its enforcement modes. Test with known benign tests (and carefully, in lab).

Validation: LKRG alerts on attempts to alter syscall handlers or kernel text sections.

2.3. Harden SELinux / AppArmor policies

Why: LSMs restrict actions of user-space processes; while a kernel-level attacker may bypass some LSMs, well-scoped policies reduce attack surface and hinder post-exploit actions.

How: Put SELinux in `enforcing`, restrict `insmod`, `finit_module`, and `call_usermodehelper` usage via policy (or restrict which users can run them), lock down writing to `/lib/modules` and to `/reptile`-like paths. Document policy exceptions.

3. Network and monitoring controls

3.1. Network IDS/behavioral detection

Why: Reptile used encrypted sessions and port 4444 + spoofed IP during exfiltration; network-level heuristics can detect unusual patterns.

Examples (Suricata rule):

```
alert tcp any any -> any 4444 (msg:"Possible Reptile C2 on 4444";
flow:established; content:"/reptile/reptile_shell"; http_client_body;
nocase; sid:1000001; rev:1;)
```

***Note**:* If the channel is custom-encrypted and not HTTP, use flow/entropy and TLS JA3 anomalies. Also alert on frequent PTR/DNS anomalies and on IP spoofing events (source IP not in expected subnets).

Validation: Replay PCAP from Report2 through Suricata and ensure rules trigger.

3.2. Endpoint detection rules (YARA) and file scanning

Why: Use the YARA rules developed in Reports 1–3 to detect static artifacts at rest and partial fragments in memory/dumps.

Example YARA: Include the `Reptile_Rootkit` YARA rule. Deploy to EDR / scanning endpoints; scan snapshots and backups.

3.3. Memory & forensic readiness

Why: As rootkits remove or hide on-disk traces, memory captures are often the only way to recover in-memory payloads and hidden processes (e.g., `/reptile_shell`). Volatility `linux.malware.check_modules` and `psscan` are proven useful in your dynamic runs.

How: Have LiME or hypervisor snapshot procedures ready to capture memory; automate Volatility analysis pipelines and keep canonical symbol sets for your kernel builds. Example: `vol -f memory.lime linux.psscan` and `vol -f memory.lime linux.malware.check_modules`.

Detection artefacts & signatures

1. High-signal YARA – deploy to EDR / file-scan

```
rule Reptile_Rootkit
{
  strings:
    $core1 = "reptile" ascii
    $core2 = "rep_mod" ascii
    $par1  = "parasite_blob" ascii
    $p2    = "/reptile/reptile_shell" ascii
    $k1    = "kallsyms_on_each_symbol" ascii
    $magik = {7F 45 4C 46}
  condition:
    uint32(0) == 0x464c457f and (any of ($core*) or 2 of ($par*))
}
```

2. Example Suricata heuristic

```
# detect high-entropy sessions on unusual ports (e.g., 4444) with small
periodic packets
alert tcp any any -> any 4444 (msg:"Suspicious encrypted session on port 4444
- possible C2"; flow:established; threshold:type both, track by_src, count 5,
seconds 60; sid:1000002; rev:1;)
```

Also create rules to flag DNS PTR anomalies, repeated small bidirectional packets, and TCP flows containing known encoded token strings seen in pcap (the sample token `cjs;y+:29%:=3%:%:8;+???+` — treat as indicator, but we may need to escape special characters in rule engines)

3. Volatility / memory indicators

- Look for hidden module names in memory: `rep_mod` flagged as `OOT_MODULE,UNSIGNED_MODULE`.
- Use `linux.psscan` to find hidden processes such as `/reptile_shell`.
- Extract kernel module list mismatch between `/proc/modules` and memory scan — treat mismatch as high severity.

Incident response playbook (containment → eradication → recovery)

Use an evidence-first approach; keep chain-of-custody notes.

Step 0 – Triage (immediate)

- Isolate host from network by disconnecting the VM's virtual NIC in VMware Workstation — do not reboot. Reboot destroys in-memory evidence.
- Take snapshots (hypervisor snapshot + disk image) and capture volatile memory with LiME or hypervisor memory dump tool.
- Collect baseline files: lsmod, ps aux, /proc/modules, dmesg -T, journalctl -k and save checksums.

Do not execute unknown binaries on the compromised host. Work on copies in an isolated analysis lab.

Step 1 — Evidence collection & quick triage

- Copy /var/log, /etc/ld.so.preload, /lib/modules/*, /reptile/* (if exists), and run YARA and hash checks.
- Capture network pcap from mirror/SPAN (if available) and isolate flows to/from 192.168.1.130 for analysis. Use existing run.pcap evidence for correlation.

Step 2 — Analysis & containment decisions

- Use Volatility to identify hidden modules/processes and dump candidate module memory for deeper analysis. Example: `vol -f memory.lime linux.malware.check_modules`.
- If kernel integrity is compromised (hidden module found), plan for full rebuild rather than attempting in-place clean. Rootkits at kernel level are not reliably removable.

Step 3 — Eradication & recovery

- Rebuild the host from trusted image (golden image) and restore data from verified backups. Rotate all credentials that may have been exposed (SSH keys, service accounts).
- Perform root cause analysis (how module was installed—vulnerability exploited, stolen credentials, or admin mistake). Patch the exploited vulnerability and implement preventive controls listed earlier. Document timelines and IOCs for the organization.

Limitations & confidence

- **Confidence:** High for behavioral assertions (runtime hooking, hidden `/reptile_shell`, C2 with encrypted sessions), because dynamic memory and pcap evidence supported these claims.
- **Limitations:** Original internet-sourced sample and compiled test differ bitwise (compiled variant added debug info). Byte-level IOCs (hashes) are therefore not interchangeable; detection should rely on behavior + robust string signatures.

Appendix — Practical demos

A. Demonstration 1 — Module signing (sign / test a module)

Goal: show how signed vs unsigned modules behave and how enforcing signature checking will prevent unsigned modules (or at least make kernel log signature failures). On BIOS systems you can sign modules and observe verification messages; to **force** rejection you either need `module.sig_enforce=1` at boot or a kernel compiled with `CONFIG_MODULE_SIG_FORCE`. If you cannot rebuild the kernel in your lab, you can still demonstrate signing + behavior and then show how to enable enforcement in principle. See kernel docs.

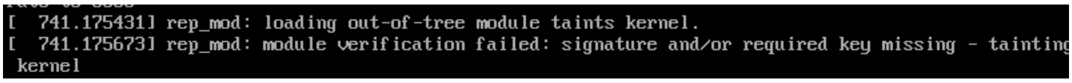
How to enforce module signature checking in CentOS7 (BIOS VM)

1) **Boot-time parameter (if kernel supports it):** add `module.sig_enforce=1` to GRUB kernel cmdline. Edit `/etc/default/grub`:

```
sudo cp /etc/default/grub /etc/default/grub.bak
sudo sed -i
's/GRUB_CMDLINE_LINUX="/GRUB_CMDLINE_LINUX="module.sig_enforce=1 /'
/etc/default/grub
sudo grub2-mkconfig -o /boot/grub2/grub.cfg
# Reboot to apply change
```

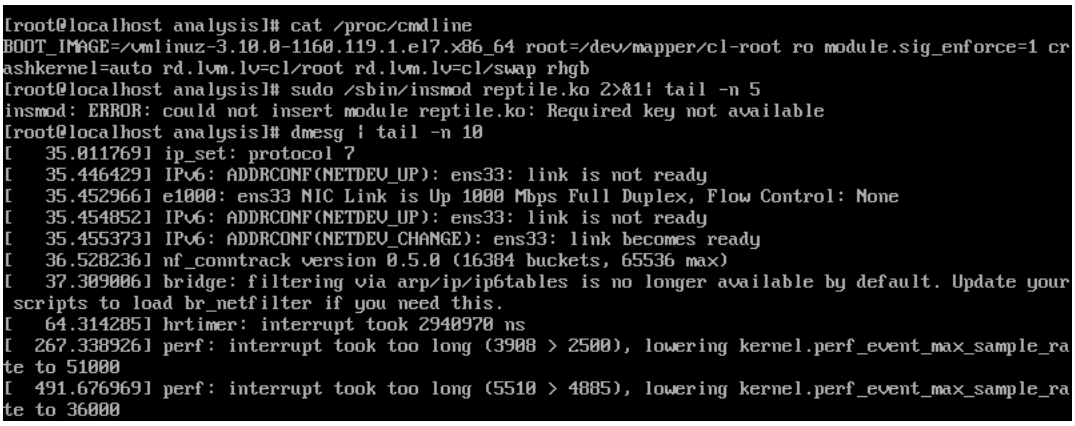
Verify:

Before enforcing:



```
[  741.175431] rep_mod: loading out-of-tree module taints kernel.
[  741.175673] rep_mod: module verification failed: signature and/or required key missing - tainting
 kernel
```

After enforcing:



```
[root@localhost analysis]# cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-3.10.0-1160.119.1.el7.x86_64 root=/dev/mapper/cl-root ro module.sig_enforce=1 cr
ashkernel=auto rd.lvm.lv=cl/root rd.lvm.lv=cl/swap rhgb
[root@localhost analysis]# sudo /sbin/insmod reptile.ko 2>&1| tail -n 5
insmod: ERROR: could not insert module reptile.ko: Required key not available
[root@localhost analysis]# dmesg | tail -n 10
[   35.011769] ip_set: protocol 7
[   35.446429] IPv6: ADDRCONF(NETDEV_UP): ens33: link is not ready
[   35.452966] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[   35.454852] IPv6: ADDRCONF(NETDEV_UP): ens33: link is not ready
[   35.455373] IPv6: ADDRCONF(NETDEV_CHANGE): ens33: link becomes ready
[   36.528236] nf_conntrack version 0.5.0 (16384 buckets, 65536 max)
[   37.309006] bridge: filtering via arp/ip/ip6tables is no longer available by default. Update your
 scripts to load br_netfilter if you need this.
[   64.314285] hrtimer: interrupt took 2940970 ns
[  267.338926] perf: interrupt took too long (3908 > 2500), lowering kernel.perf_event_max_sample_ra
te to 51000
[  491.676969] perf: interrupt took too long (5510 > 4885), lowering kernel.perf_event_max_sample_ra
te to 36000
```

Expected: kernel refuses to load unsigned modules if key missing.

2) **Rebuild kernel with CONFIG_MODULE_SIG_FORCE=y** – stronger, permanent enforcement (recommended for production hardened images). This requires building a kernel image and installing it; follow distro kernel building docs (out of scope for short lab, but mention in report as recommended).
Caveat (BIOS VM): UEFI/MOK enrollment workflows (mokutil) are not applicable in BIOS mode — there is no UEFI Secure Boot. However, `module.sig_enforce=1` or a kernel compiled with `CONFIG_MODULE_SIG_FORCE` still enforces signature checking at kernel level (no UEFI required) if the kernel has the public keys available in its keyring — see kernel admin guide.

B. Demonstration 2 — Disable dynamic module loading (quick mitigation)

Goal: show a fast, reversible way to block *new* kernel modules in a running kernel (useful for containment).

Commands:

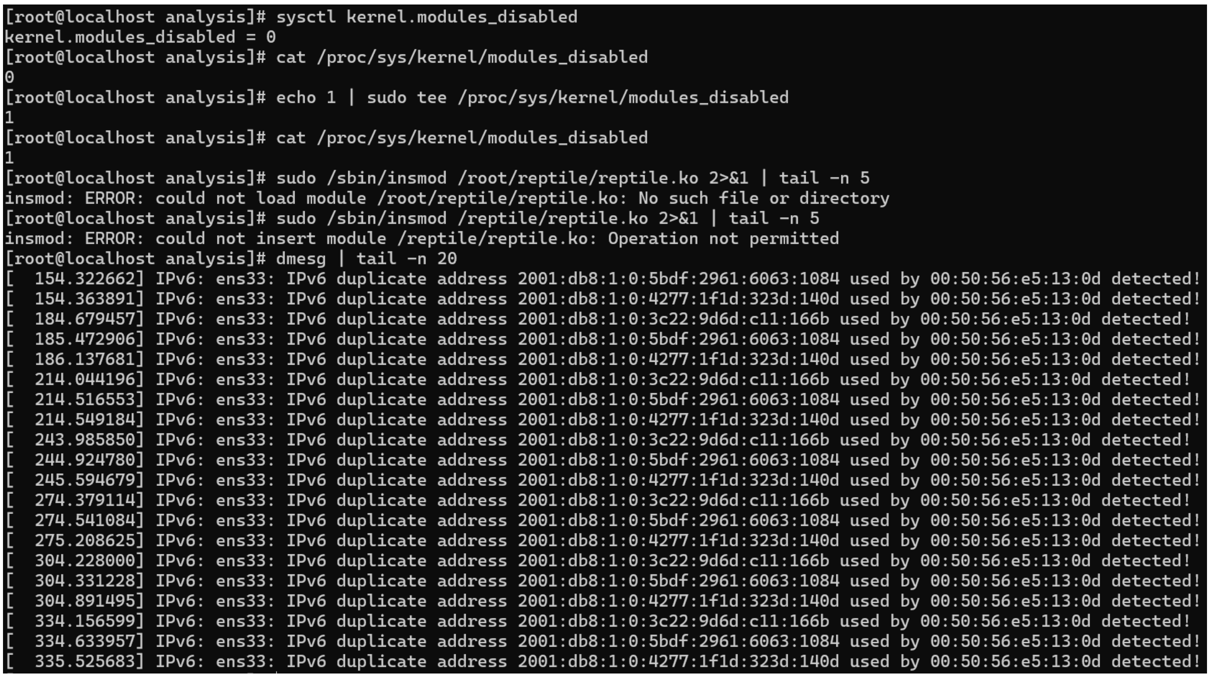
```
# View current setting
sysctl kernel.modules_disabled
# or
cat /proc/sys/kernel/modules_disabled

# Disable module loading until next reboot
echo 1 | sudo tee /proc/sys/kernel/modules_disabled

# Verify
cat /proc/sys/kernel/modules_disabled   # should output 1

# Try to insmod a module (should fail)
sudo /sbin/insmod /root/reptile/reptile.ko 2>&1 | tail -n 5
dmesg | tail -n 20
```

Result:



```
[root@localhost analysis]# sysctl kernel.modules_disabled
kernel.modules_disabled = 0
[root@localhost analysis]# cat /proc/sys/kernel/modules_disabled
0
[root@localhost analysis]# echo 1 | sudo tee /proc/sys/kernel/modules_disabled
1
[root@localhost analysis]# cat /proc/sys/kernel/modules_disabled
1
[root@localhost analysis]# sudo /sbin/insmod /root/reptile/reptile.ko 2>&1 | tail -n 5
insmod: ERROR: could not load module /root/reptile/reptile.ko: No such file or directory
[root@localhost analysis]# sudo /sbin/insmod /reptile/reptile.ko 2>&1 | tail -n 5
insmod: ERROR: could not insert module /reptile/reptile.ko: Operation not permitted
[root@localhost analysis]# dmesg | tail -n 20
[  154.322662] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  154.363891] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
[  184.679457] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:3c22:9d6d:c11:166b used by 00:50:56:e5:13:0d detected!
[  185.472906] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  186.137681] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
[  214.044196] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:3c22:9d6d:c11:166b used by 00:50:56:e5:13:0d detected!
[  214.516553] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  214.549184] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
[  243.985850] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:3c22:9d6d:c11:166b used by 00:50:56:e5:13:0d detected!
[  244.924780] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  245.594679] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
[  274.379114] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:3c22:9d6d:c11:166b used by 00:50:56:e5:13:0d detected!
[  274.541084] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  275.208625] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
[  304.228000] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:3c22:9d6d:c11:166b used by 00:50:56:e5:13:0d detected!
[  304.331228] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  304.891495] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
[  334.156599] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:3c22:9d6d:c11:166b used by 00:50:56:e5:13:0d detected!
[  334.633957] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:5bdf:2961:6063:1084 used by 00:50:56:e5:13:0d detected!
[  335.525683] IPv6: ens33: IPv6 duplicate address 2001:db8:1:0:4277:1f1d:323d:140d used by 00:50:56:e5:13:0d detected!
```

Expected: `insmod` fails; dmesg indicates module load refusal. This setting cannot be reverted until reboot — useful for immediate containment. See Linux articles and practical notes

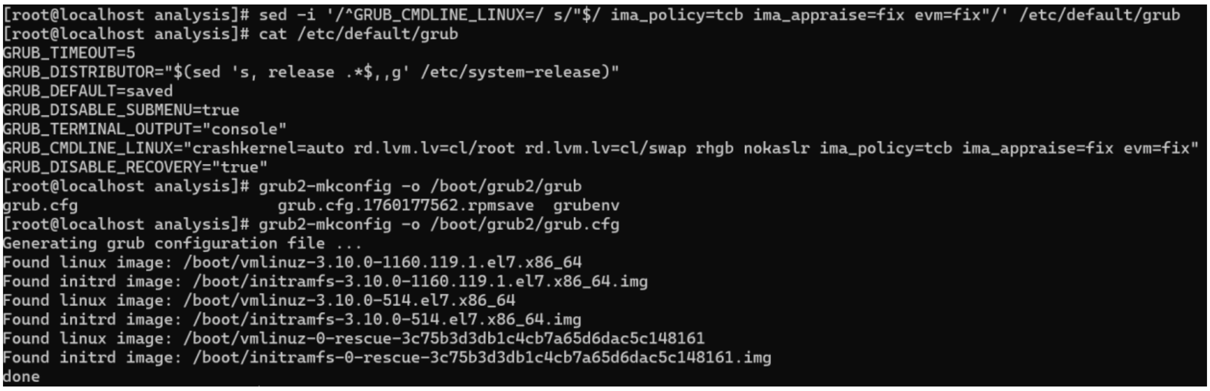
Recommendation: use this when we suspect kernel compromise and we want to prevent further module insertion while we investigate.

C. Demonstration 3 — IMA / EVM (runtime file/module integrity enforcement) (for UEFI Secure boot)

Goal: enable IMA/EVM appraisal and show how it prevents execution of tampered files or unapproved modules.

1) Add kernel cmdline policies (GRUB)

```
# Edit /etc/default/grub and append to GRUB_CMDLINE_LINUX
sudo sed -i '/^GRUB_CMDLINE_LINUX=/ s/"$/ ima_policy=tcb ima_appraise=fix evm=fix"/' /etc/default/grub
sudo grub2-mkconfig -o /boot/grub2/grub.cfg
sudo reboot
```



```
[root@localhost analysis]# sed -i '/^GRUB_CMDLINE_LINUX=/ s/"$/ ima_policy=tcb ima_appraise=fix evm=fix"/' /etc/default/grub
[root@localhost analysis]# cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto rd.lvm.lv=cl/root rd.lvm.lv=cl/swap rhgb nokaslr ima_policy=tcb ima_appraise=fix evm=fix"
GRUB_DISABLE_RECOVERY="true"
[root@localhost analysis]# grub2-mkconfig -o /boot/grub2/grub
grub.cfg                     grub.cfg.1760177562.rpmsave  grubenv
[root@localhost analysis]# grub2-mkconfig -o /boot/grub2/grub.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.10.0-1160.119.1.el7.x86_64
Found initrd image: /boot/initramfs-3.10.0-1160.119.1.el7.x86_64.img
Found linux image: /boot/vmlinuz-3.10.0-514.el7.x86_64
Found initrd image: /boot/initramfs-3.10.0-514.el7.x86_64.img
Found linux image: /boot/vmlinuz-0-rescue-3c75b3d3db1c4cb7a65d6dac5c148161
Found initrd image: /boot/initramfs-0-rescue-3c75b3d3db1c4cb7a65d6dac5c148161.img
done
```

2) After reboot — install user tools and prepare policy

```
sudo yum install -y ima-evm-utils
# Generate policy and set xattrs on files that will be appraised (this step
can be time-consuming)
sudo evmctl --help   # read the tool usage for your workflow
# Example: appraise a single file
sudo evmctl ima_sign --key MOK.priv --cert MOK.pem --hash sha256
/usr/bin/somebinary
```

Expected validation:

- `dmesg` should show IMA messages on appraisal.
- Attempt to run or load a module that does not pass appraisal should be blocked or logged depending on policy.

Caveat & notes: full deployment of IMA/EVM is non-trivial: you must generate and record IMA labels (xattrs) for all files before enabling appraisal otherwise legitimate files are blocked. See Red Hat guide for step-by-step relabel process.

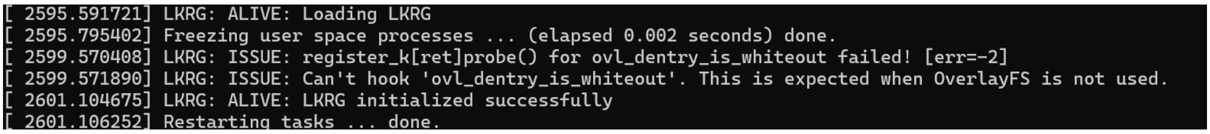
D. Demonstration 4 — Install & test LKRG (kernel runtime integrity monitor)

Goal: deploy LKRG and observe detection of kernel modifications such as syscall table hooking (the behavior Reptile uses).

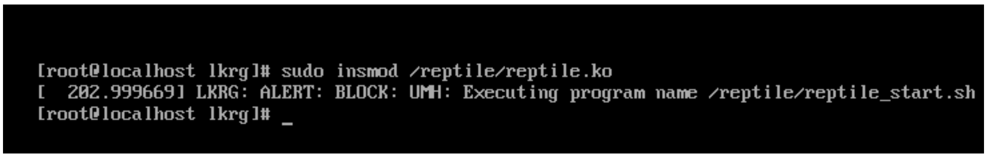
```
sudo yum install -y git gcc make kernel-devel-$(uname -r)
git clone https://github.com/lkrg-org/lkrg.git /root/lkrg
cd /root/lkrg
make
sudo make install
```

Verify:

```
sudo insmod ./lkrg.ko
dmesg | tail -n 20
```



```
[ 2595.591721] LKRG: ALIVE: Loading LKRG
[ 2595.795402] Freezing user space processes ... (elapsed 0.002 seconds) done.
[ 2599.570408] LKRG: ISSUE: register_k[ret]probe() for ovl_dentry_is_whiteout failed! [err=-2]
[ 2599.571890] LKRG: ISSUE: Can't hook 'ovl_dentry_is_whiteout'. This is expected when OverlayFS is not used.
[ 2601.104675] LKRG: ALIVE: LKRG initialized successfully
[ 2601.106252] Restarting tasks ... done.
```



```
[root@localhost lkrg]# sudo insmod /reptile/reptile.ko
[  202.999669] LKRG: ALERT: BLOCK: UMH: Executing program name /reptile/reptile_start.sh
[root@localhost lkrg]# _
```

Expected: LKRG messages in dmesg when modifications detected. If you see LKRG warnings, you demonstrated successful detection of in-memory tampering.

Caveat: LKRG itself is a kernel module; in environments with strict signing enforcement you must sign/build LKRG accordingly. See LKRG docs for DKMS installation and kernel compatibility.

E. Demonstration 5 — Network detection (Suricata) — replay your lab pcap

Goal: replay the `run_capture.pcap` you used in Report2 and show Suricata alerts for rules detecting C2 or encoded payload tokens.

Prerequisite: install Suricata and put a simple rule into `local.rules`.

```
sudo yum install -y epel-release
sudo yum install -y suricata
```

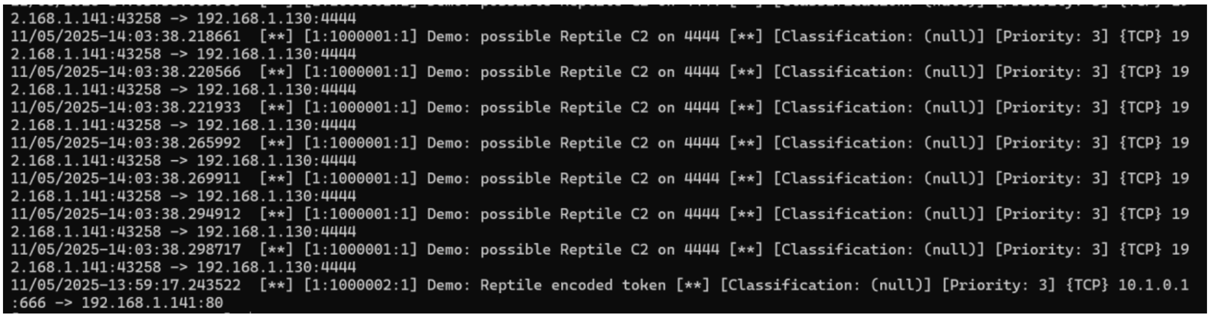
Edit `/etc/suricata/rules/local.rules` and paste:

```
alert tcp any any -> any 4444 (msg:"Demo: possible Reptile C2 on 4444";
flow:established; sid:1000001; rev:1;)
# If the pcap contains the encoded token 'cjs;y' include content match (escape
as needed)
alert tcp any any -> any any (msg:"Demo: Reptile encoded token";
content:"cjs;y"; sid:1000002; rev:1;)
```

Run Suricata against the pcap:

```
sudo suricata -r /path/to/run_capture.pcap -c /etc/suricata/suricata.yaml -S
/etc/suricata/rules/local.rules --runmode=single
# Check alerts:
sudo tail -n 50 /var/log/suricata/alerts.log
```

Result:



```
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.218661  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.220566  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.221933  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.265992  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.269911  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.294912  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-14:03:38.298717  [**] [1:1000001:1] Demo: possible Reptile C2 on 4444 [**] [Classification: (null)] [Priority: 3] {TCP} 19
2.168.1.141:43258 -> 192.168.1.130:4444
11/05/2025-13:59:17.243522  [**] [1:1000002:1] Demo: Reptile encoded token [**] [Classification: (null)] [Priority: 3] {TCP} 10.1.0.1
:666 -> 192.168.1.141:80
```

Expected: Suricata will generate alerts for flows on port 4444 and/or for the specified content `cjs;y`. This proves the network detection rules can pick up the lab C2 traffic you observed in Report2. See Suricata blogs/guides for tuning and false-positive control.

References

Internal project files:

- Report 1 — Malware Classification.
- Report 2 — Dynamic Analysis.
- Report 3 — Static Analysis.

Authoritative external sources:

1. The Linux Kernel Organization. **Kernel module signing** [online]. Portland (OR): The Linux Kernel Organization; 2018. Available from: https://www.kernel.org/doc/html/latest/admin-guide/module-signing.html [Accessed: 6 Nov 2025].
2. The Linux Kernel Organization. **The Linux Kernel documentation** [online]. Portland (OR): The Linux Kernel Organization; 2020. Available from: https://www.kernel.org/doc/html/latest/ [Accessed: 6 Nov 2025].
3. Red Hat, Inc. **Enhancing security with the kernel integrity subsystem** [online]. Raleigh (NC): Red Hat; 2021. Available from: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/kernel_administration_guide/enhancing_security_with_the_kernel_integrity_subsystem [Accessed: 6 Nov 2025].
4. Red Hat, Inc. **How to use the Linux kernel's Integrity Measurement Architecture (IMA)** [online]. Raleigh (NC): Red Hat; 2020. Available from: https://www.redhat.com/en/blog/how-use-linux-kernels-integrity-measurement-architecture [Accessed: 6 Nov 2025].
5. lkrg-org. **lkrg: Linux Kernel Runtime Guard** [online]. GitHub repository; 2023. Available from: https://github.com/lkrg-org/lkrg [Accessed: 6 Nov 2025].
6. LKRG Project. **Linux Kernel Runtime Guard — Official site** [online]. [Place unknown]: LKRG Project; 2023. Available from: https://lkrg.org/ [Accessed: 6 Nov 2025].
7. 504ensicsLabs. **LiME — Linux Memory Extractor** [online]. GitHub repository; 2014. Available from: https://github.com/504ensicsLabs/LiME [Accessed: 6 Nov 2025].
8. Volatility Foundation. **Volatility 3 — Documentation and plugins** [online]. [Place unknown]: Volatility Foundation; 2021. Available from: https://volatility3.readthedocs.io/en/latest/ [Accessed: 6 Nov 2025].
9. Volatility Foundation. **Volatility Framework** [online]. [Place unknown]: Volatility Foundation; 2021. Available from: https://volatilityfoundation.org/the-volatility-framework/ [Accessed: 6 Nov 2025].
10. OISF / OMD. **Suricata — Rules documentation** [online]. [Place unknown]: OISF / Suricata project; 2024. Available from: https://docs.suricata.io/en/latest/rules/index.html [Accessed: 6 Nov 2025].
11. AhnLab Security Emergency response Center (ASEC). **Reptile malware targeting Linux systems** [online]. Seoul: AhnLab; 2023. Available from: https://asec.ahnlab.com/en/55785/ [Accessed: 6 Nov 2025].
12. Mandiant / Google Cloud Threat Intelligence. **Cloaked and Covert: Uncovering UNC3886 espionage operations** [online]. Mountain View (CA): Google Cloud; 2024.

13. Available from: https://cloud.google.com/blog/topics/threat-intelligence/uncovering-unc3886-espionage-operations [Accessed: 6 Nov 2025].
14. U.S. Department of Defense / NSA & FBI (joint advisory). **Cybersecurity Advisory: Russian GRU 85th GTsSS Deploys Previously Undisclosed Drovorub Malware** [online]. Washington (DC): U.S. Department of Defense; 2020-08-13. Available from: https://media.defense.gov/2020/aug/13/2002476465/-1/-1/0/CSA_drovorub_russian_gru_malware_aug_2020.pdf [Accessed: 6 Nov 2025].
15. Exploit Database. **Reptile rootkit — reptile_cmd / exploit module** [online]. [Place unknown]: Exploit-DB; 2019. Available from: https://www.exploit-db.com/exploits/47804 [Accessed: 6 Nov 2025].
16. DFIR.ch. **Reptile's custom kernel-module launcher — analysis** [online]. Zurich: dfir.ch; 2024. Available from: https://dfir.ch/posts/reptile_launcher/ [Accessed: 6 Nov 2025].
17. The Hacker News. **Reptile Rootkit: Advanced Linux Malware Targeting South Korean Systems** [online]. New York: The Hacker News; 2023-08-05. Available from: https://thehackernews.com/2023/08/reptile-rootkit-advanced-linux-malware.html [Accessed: 6 Nov 2025].
18. Threat research / advisories summary pages (various). **Reptile — community threat pages and indicators** [online]. Multiple publishers (Wiz, Threat actors advisories, vendor blogs); 2023–2025. Representative pages: https://threats.wiz.io/all-tools/reptile, https://otx.alienvault.com/pulse/64d267daf70e411d6f941487 [Accessed: 6 Nov 2025].
19. Kernel project / distribution documentation. **Kernel lockdown and Secure Boot (guidance)** [online]. The Linux Kernel Organization / upstream docs; 2021. Available from: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html and vendor docs on Secure Boot; [Accessed: 6 Nov 2025].
20. OpenEuler Project. **Kernel module signing — documentation** [online]. Beijing: openEuler; 2022. Available from: https://docs.openeuler.org/en/docs/22.09/docs/ShangMi/kernel-module-signing.html [Accessed: 6 Nov 2025].
21. Red Hat, Inc. **Managing, monitoring and updating the kernel — Enhancing security with the kernel integrity subsystem (RHEL 9)** [online]. Raleigh (NC): Red Hat; 2023. Available from: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/managing_monitoring_and_updating_the_kernel/enhancing-security-with-the-kernel-integrity-subsystem_managing-monitoring-and-updating-the-kernel [Accessed: 6 Nov 2025].

---END---