Linux kernel remote logging: approaches, challenges, implementation

A talk by

Solar Designer <solar@openwall.com>
@solardiz

@Openwall / https://www.openwall.com

Credits: LKRG, Binarly, CIQ/Rocky Linux

March 1, 2024
Zagreb, Croatia

Why remote logging

* Troubleshooting and post-mortem analyses of (non-)security incidents
  + System's local logs might be unavailable, incomplete, or tampered with
  + There's commonly no way to know the local logs are complete and intact
* Centralized processing for SIEM, EDR, ...
* Compliance

Pre-existing remote logging solutions

* syslogd and its protocol - many implementations with varying features

* CORE-SDI ssyslog/msyslog - blockchain before it was cool, logging to SQL

* Linux kernel netconsole
  + Uses syslog protocol

* rsyslog Reliable Event Logging Protocol (RELP)
  + Also supported in rsyslog's librelp, including over TLS since 2013

* NXLog
  + Not obviously Open Source - community edition under custom license

* systemd can export/import its journal to a remote node over HTTPS POSTs

syslog

* De-facto standard on Unix and beyond since 1980s
  + Plaintext over Unix domain socket or UDP
* Specified in RFCs in 2000s
* Wikipedia lists the below:

The BSD syslog Protocol. RFC 3164 (obsoleted by RFC 5424)
Reliable Delivery for syslog. RFC 3195
The Syslog Protocol. RFC 5424
TLS Transport Mapping for Syslog. RFC 5425
Transmission of Syslog Messages over UDP. RFC 5426
Textual Conventions for Syslog Management. RFC 5427
Signed Syslog Messages. RFC 5848
Datagram Transport Layer Security (DTLS) Transport Mapping for Syslog. RFC 6012
Transmission of Syslog Messages over TCP. RFC 6587

CORE-SDI ssyslog/msyslog as announced on Bugtraq in 1998

```
                       DO YOU TRUST YOUR SYSTEM'S LOGS?

                           Secure System Logging
                      ** FREE SOURCE CODE AVAILABLE **


        CORE SDI S.A. introduces a new cryptographically secure system
logging tool.

        SECURE SYSLOG (ssyslog) is available for UNIX systems.  Designed to
replace the syslog daemon, ssyslog implements a cryptographic protocol called
PEO-1 that allows the remote auditing of system logs. Auditing remains possible
even if an intruder gains superuser privileges in the system, the protocol
guarantees that the information logged before and during the intrusion process
cannot be modified without the auditor (on a remote, trusted host) noticing.
```

Linux kernel netconsole

* A standard feature of the Linux kernel

* Can be built into the kernel image, but distros usually build as a module

* Sends kernel messages via syslog protocol over UDP
  + Unreliable and insecure transport - but simple and thus reliable setup

* Specialized and low-level enough that it:
  + Starts network logging early and doesn't fail even on a kernel panic
  + Requires Ethernet and manual specification of the target MAC address

* Intended to work over local Ethernet only, but also works over the Internet
  + Just specify the gateway's MAC address along with the target's IP address

systemd journal export

* A standard feature on modern Linux distros that use systemd
  + but not available on those that don't

* Everything is logged locally first
  + Pro: can send previously unsent pre-reboot logs after a reboot
  + Con: cannot send what's never written (e.g., on kernel panic or disk full)

* Exporting only kernel messages requires tricky changes or wrapper scripts
  + "journal: add concept of "journal namespaces" #14178" might help

* Client authentication not yet fully implemented?
  + "systemd-journal-remote using HTTPS & TrustedCertificateFile neither requires nor verifies client certificates #4092" open since 2016
  + "journal-remote: bugfix to re-enable ssl key check #10707" partial? fix?

Linux Kernel Runtime Guard

* Project of Adam 'pi3' Zabrocki
  + Brought under Openwall umbrella for its first public release in 2018
  + Post-detection of (and response to) kernel rootkits and exploits
    + Openwall's most controversial project ever? beats even John the Ripper?

* Linux kernel module that performs
  + Runtime integrity checking of the kernel
  + Detection of security vulnerability exploits against the kernel

* Delivery, storage, and processing of LKRG security events to/on a remote system is a natural extension of LKRG's functionality
  + In their basic form, the events are just kernel log messages
  + In an advanced form, they may also include remote-only messages and blobs (e.g., kernel module, eBPF, and exploit program binaries or their hashes)

Protocol security and operational needs and wishes

* Transport security (TLS-alike) - mandatory, with directionality options:
  + Strictly one-way (no two-way handshake => no forward secrecy, replay risks, no or blind retransmits at this layer)
  + Two-way for handshake and acks only, one-way for actual messages
  + Two-way also for control messages pulled from server (for managed response)
  + Two-way also with ability to push control messages by server (ditto)

* Long-term encryption and authentication of messages and blobs - optional
  + No forward secrecy at this layer
  + Handy to have pass-through security via proxy/storage/relay servers

* Not too susceptible to (D)DoS
  + Even more importantly, not a traffic amplification oracle

Protocol operational needs and wishes

* Handle concurrency (not mix up pieces of concurrent messages)

* "Reliable" delivery (queueing, retransmits)

* Congestion control (just use TCP or/and rate-limiting)

* Message prioritization ("kernel panic" sent out-of-band before pending blobs, a reason to use UDP or more than one TCP connection)

* "Roaming" support (one-way vs. needing to re-handshake vs. sharing state)

* Message encapsulation layer (custom extensible, Cap'n Proto, protobuf)

Transport security protocol options

```
* Noise protocol framework
  + Widely accepted by security community (WireGuard in Linux kernel, etc.)
  + Flexible (many handshake pattern options, crypto primitives agility)
  + One Noise instantiation (pattern, primitives) can be not too complex

* TLS
  + Industry standard
  + Complex - "not our problem" if we use a library and keep it updated
    + There will be non-updated systems, so this merely shifts responsibility
  + Brings one-way option mostly out of consideration
  + Brings doing it 100% from kernel mostly out of consideration
    + Already exists in upstream kernel, but not always built in distros
  + Handshake in userspace, message sending from kernel still an option
```

Transport security protocol building blocks

```
* NaCl (pronounced "salt") provides DJB primitives

* Noise protocol implementations, with choice between:
  + Established DJB primitives (Curve25519, ChaCha20, Poly1305)
    + libhydrogen v0 (Noise proper), Monocypher-Handshake (Noise-alike)
  + Legacy DJB primitives (Curve25519, Salsa20, Poly1305)
    + Original NaCl, TweetNacl, libsodium
  + Newer DJB primitives (Curve25519, Gimli)
    + libhydrogen (Noise proper with Gimli)
  + NIST primitives (NIST curves, AES, SHA-512)
    + Weird combination, AES in software is cache-timing-unsafe

* TLS implementations
  + Also choice of primitives, and NIST not so weird
```

Noise handshake patterns

* Noise can optionally authenticate the parties by their static keys

* Noise has both one-way (non-)handshake and two-way handshake patterns

* We can choose a two-way pattern usable as one-way on no/before response

* If a response never arrives, we can be restarting the handshake for each message indefinitely (include payload in "first" messages)
  + Operating like a one-way pattern would

* One such pattern is "IK"
  + This is the one used by WireGuard
  + Unfortunately, not readily in libhydrogen

Messages vs. stream

* Noise is message-oriented

* Noise over UDP is a perfect fit (one datagram at a time, size in header)

* Noise over TCP is trickier (receive stream, but decrypt individual messages), our options are:
  + Introduce plaintext header with message length field (followed by Noise message of that length)
  + Alternate encrypted metadata (containing next message's length) and actual messages (both Noise)
    + Yet the individual message lengths would often be exposed via separate TCP segments
  + Send fixed-size messages with actual length in their header (encrypted)
    + Wasteful, but also mitigates traffic analysis

Message types or/and numbers

* With Noise over TCP, we just do a Noise handshake on (re)connect
  + The handshake messages arrive in order
  + There's no point in redoing a handshake within a TCP connection

* With Noise over UDP, we need to distinguish handshake vs. data messages
  + Messages can arrive out of order
  + Handshake can be restarted at any time without any other indication
  + With multi-message handshake patterns, we need to distinguish the steps
  + We can introduce a header containing message type or/and number
    + Alternatively, we can ensure each type has its distinct size range
  + libhydrogen has built-in support for message numbers, but that's to fail decryption of unexpected messages
    + Not what we need here, but useful against replay within a TCP stream

Implementation plan

* Proceed with a trial implementation of a Noise protocol with DJB primitives
  + Pros: not too complex yet not custom, security community's choice, agile
  + Cons: possibly not as "serious-sounding" to some users as TLS+NIST would be

* Choose Gimli over ChaCha20+Poly1305
  + Pros: newer, smaller, all-in-one, peer-reviewed, readily in libhydrogen
  + Cons: newer so maybe not as peer-reviewed as ChaCha20+Poly1305 yet

* Start with the trivial Noise pattern "N" and without an encapsulation layer

* Start with either TCP or UDP depending on implementation constraints
  + Noise lets us add or switch to the other later

* In the end, proceeded with TCP and plaintext Noise message length headers

More decisions were yet to make

* Transport security is just one (important) tip of the iceberg

* Many non-trivial design and implementation decisions to make in other areas
  + Concurrency
  + Recursion
  + Locking
    + Deadlock and priority inversion risks
  + Queueing
  + Failure handling

* Start simple (simplified), then revise and extend to deal with issues

When and where to send from

Messages can be (attempted to be) sent to the remote...

```
* Right upon the corresponding event (same places where we currently printk)
  + directly from LKRG's code
  + or from an extra console we'd register with the kernel
  + or from a hackish hook into printk

* From a pre-existing work queue (runs from a pre-existing kworker thread)
* From an own kthread (that we'd spawn)
* From userspace
  + From a service process we'd spawn/manage externally or by/from the kernel
    + fork_usermode_driver on Linux 5.9+, fork_usermode_blob before
```

Send right away? Not so easy.

* Execution context can vary - task, hw interrupt, sw interrupt ("bottom half")
  + LKRG's kprobes are usually optimized to ftrace, but when they are not we're
    in INT3 handler

* Network stack APIs might not be safely usable in the current context
  + Tried "in_interrupt() ? buffer_and_send_later() : send_right_away()"
    + Too often true and wasn't sufficient to consistently avoid issues

* Sending could delay further processing
  + Even LKRG's security response such as killing a task could be delayed
    + Not strictly a new problem: some existing console drivers could do it
    + As a separate sub-project: need to rework logging vs. response?

* Recursion is a concern (trying to send a message can generate an event)

Linux printk recursion

* Within our supported kernel versions, Linux switched from almost disallowing printk recursion to allowing up to 3 levels of recursion

* Linux has tricky custom logic to detect recursion

Old:

```
                 * If a crash is occurring during printk() on this CPU,
                 * then try to get the crash message out but make sure
                 * we can't deadlock. Otherwise just return to avoid the
                 * recursion and return - but flag the recursion so that
                 * it can be printed at the next appropriate moment:
```

New:

```
#define PRINTK_MAX_RECURSION 3
```

Potential deadlock

* When sending right away (literal recursion)

```
        lock_net(); /* for our socket, etc. */
        send();     /* generates another event, recurses to the above */
```

* When sending buffered events e.g. from a work queue (not recursion)

```
        lock_buf(); /* for reading from and then emptying the buffer */
        send();     /* generates another event, reaches buffering, so: */
        lock_buf(); /* elsewhere in our code tree, for adding to the buffer */
```

* Solution (kind of): use a trylock before reading buffer, postpone if locked

Priority inversion

* Another reason why we'd buffer and send later is to avoid delaying further processing in our current code path
  + which is potentially timing sensitive and high-priority

* However, we could end up waiting anyway - on the buffer lock
  + which is acquired by our lower-priority work queue / kthread / syscall

* Solution (kind of): acquire the buffer lock only briefly, to copy the buffer content (full or a portion) to a local buffer, release the lock, send the local buffer, re-acquire the lock if there's more to send
  + Linux kernel's own printk vs. syslog(2) similarly releases the lock during copy_to_user, then re-acquires
  + Issue: what to do if the sending fails? Re-add to buffer? That's potential re-ordering, and what if buffer full by then?

What kind of buffer(s)

* Solution (kind of): a ring of separate buffers with their separate locks

* Linux switched from simpler to complex prb (for "printk ring buffer") within our supported versions
  + So we can't easily plug into and reuse their buffer internals, but we can reuse the concepts - maybe later

* Meanwhile, we can access the kernel's existing ring buffer (whatever it is) via the high-level devkmsg_read
  + Intended for userspace, ends in a copy_to_user
  + If we don't have a valid __user pointer, we could let this EFAULT, then extract from file->private_data

Send/buffer from a "console"? Pros.

Successful experiment: instead of adding code to LKRG's own p_print_log, register an extra console with the kernel and send from there. Pros:

* Can capture/send all kernel messages, not only LKRG's (we can filter)

* Usable separately from LKRG (the experiment was a separate kernel module), and provides functionality useful even without LKRG (e.g., for system troubleshooting - similar to netconsole, but not-so-low-level and with a transport security layer we can add, which netconsole lacks)

* We don't need to provide our own buffer for printf-formatting a kernel message - we just printk with format like LKRG did so far

* Decoupling of LKRG code from the sending code

Send/buffer from a "console"? Cons.

* Other parts of the kernel, modules, and maybe even userspace root can spoof kernel messages and we'd have a hard time extracting genuine LKRG messages

* The decoupling is not good enough to eliminate the need for further buffering and postponed processing

* Greater risk of recursion if we send more than LKRG's own messages and do so right away (can avoid that)

* Obeys console log level setting, which is a useful feature for a generic remote console, but it's an extra way to silence LKRG and besides we could want to keep LKRG's "ALIVE" messages hidden from local consoles (remote-only)

* We'll also be sending something from elsewhere in LKRG long-term anyway

When and where to send from - decision

* Do register a custom console, but don't send right from there and don't use the message supplied to the handler
  + Instead, merely queue a "work" to run from a pre-existing kworker thread

* Also, queue the same kind of "work" from LKRG's own code when it knows it's just logged a message (redundant)

* In the queued "work", trylock our socket, read from the kernel message buffer via the high-level API, encrypt, and send (if necessary, also [re]connect)

To reiterate, other decisions so far:

* Use libhydrogen to implement Noise, initially the trivial pattern "N"
* Use TCP and plaintext Noise message length headers

Initial implementation

* Released in LKRG 0.9.8 on February 28, 2024
  + Along with builds for Rocky Enterprise Linux 8.9 and 9.3 via SIG/Security

* Logs not only messages generated by LKRG, but also all other kernel messages

* The sending component is in the LKRG kernel module itself
  + Communication is over a TCP socket opened write-only, which limits LKRG's remote attack surface - great
  + Write-only is possible due to use of the "N" pattern, which unfortunately precludes implementation of forward secrecy - not great

* The receiving and logging counterpart is in a userspace daemon, lkrg-logger

* There are also additional userspace utilities, lkrg-keygen and lkrg-logctl

Limitations

* Replay protection is partial - messages from the middle of a TCP connection cannot be replayed on their own (won't be accepted), but an entire TCP connection or its starting portion can be
  + Something to improve if we can (the one-way communication limits this)

* There's no explicit server authentication, however security against a spoofed/MITM server is achieved through only encrypting to the correct server's pre-configured public key
  + May be fine as-is (and is also a side-effect of one-way communication)

* There's currently no explicit client authentication
  + A major shortcoming to be addressed
  + Only clients with knowledge of the server's public key can send messages that would be accepted by the server, but that's not proper authentication

Hindsight and the future

* The current implementation achieves quite little over being in userspace
  + Not having a userspace sender process, it can't get OOM-killed, etc.

* Our use of Linux kernel's networking APIs is at a higher level than netconsole's, so that configuration is easier and not tied to Ethernet

* Unfortunately, it's also too high-level to work after a kernel panic, which would have been a good reason to have this in the kernel
  + "Net: Implement deferred panic #314" is a workaround idea

* Many ideas are not yet implemented - let's see how the initial release is received and used, what's actually in demand, and maybe what's contributed

* Remote logging functionality may also be made available separately from LKRG

Contact information and credits

e-mail
Solar Designer <solar@openwall.com>

Twitter
@solardiz @Openwall

websites
https://www.openwall.com https://lkrg.org

This research and initial implementation in LKRG have been sponsored by Binarly
software supply chain security platform https://binarly.io

The public release, Rocky Linux integration, and this talk are due to my work
at CIQ, the primary corporate sponsor of Rocky Linux https://ciq.com