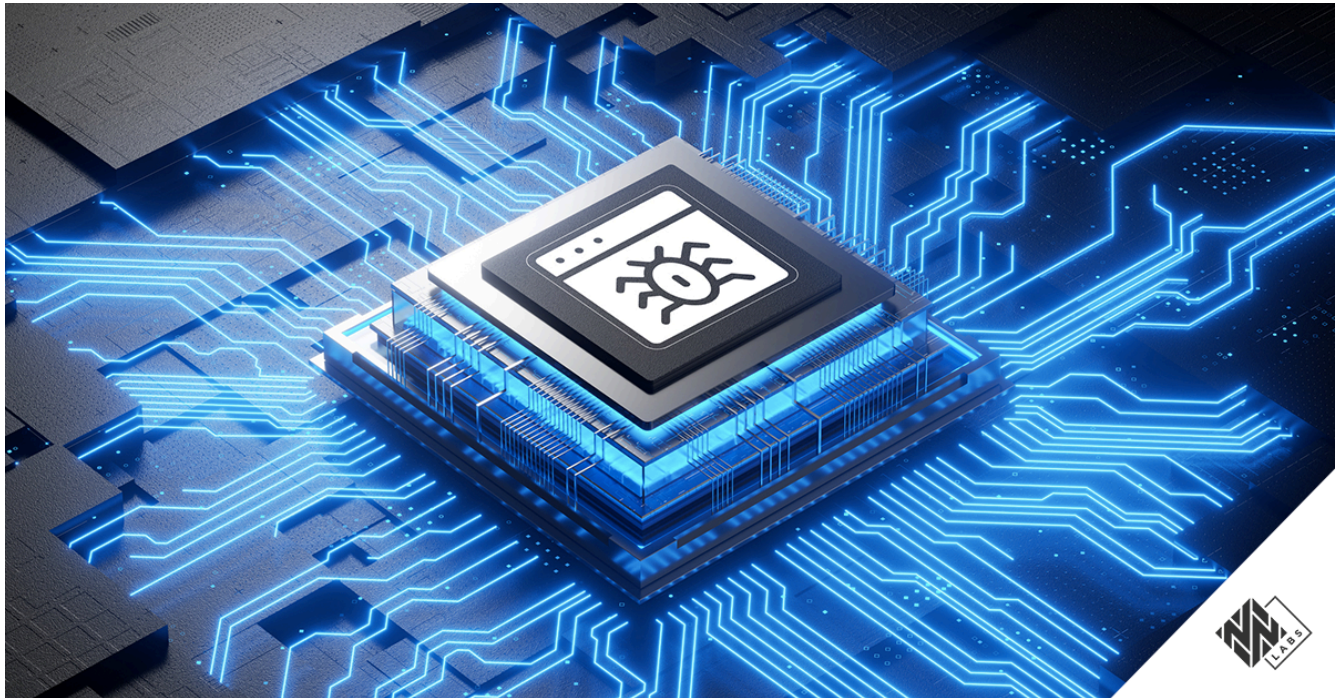


KATARU: IoT Malware Adopts Public LPE Exploits

 nozominetworks.com/blog/kataru-iot-malware-adopts-public-lpe-exploits

Nozomi Networks

September 10, 2026



KATARU is an IoT malware sample observed in a recent honeypot compromise through Telnet credential brute forcing. While it retains familiar Mirai-style botnet functionality, it stands out for its unusually broad capability set, including multiple Linux n-day local privilege escalation exploits, extensive persistence coverage across Linux and embedded environments, encrypted C2 communications, anti-analysis checks and decoy traffic. Several implementation artifacts, including architecture-mismatched exploit code, RFC-derived test values, and broad cross-platform persistence logic, strongly suggest that KATARU was assembled with AI assistance rather than developed and validated as a mature malware family. This post walks through the infection chain, key technical behaviors and defensive recommendations for identifying and reducing exposure to similar IoT threats.

Technical Details of KATARU

Initial access

The infection was observed in August after Telnet credential brute forcing against one of our honeypots from an IP address in Vietnam. After access was obtained, the actor used

BusyBox commands to retrieve and execute an ARM payload saved as vlxx.arm (13382c16e2401b07451577b46e634b8031ec254d98b876e59692b5fa22abc1d4).

The command sequence and markers observed during staging, including condi72 and condixx, indicate that the payload was likely delivered by loader cc76bc218627279ecb4d0ce74ad2651e9db9e3e843e35d6569576e056e3a9218 or a closely related variant. Source code for earlier versions of this rudimentary dropper has previously appeared online.

```
{
  "line": "{\"record_id\": \"6b280c13-5276-4888-9189-b65bb8e7f78\", \"honeypot\": \"telnetiohoneypot\", \"type\": \"command\", \"ts\": \"2026-08-06 12:04:37\", \"protocol\": \"telnet\", \"attacker_ip\": \"160.191.242.92\", \"country\": \"VN\", \"command\": \"./condi72 telnet.arm; /bin/busybox xxdnoc\"\", \"timestamp\": \"1726022345800000000\", \"date\": \"2026-08-06T13:19:05.000\", \"fields\": { \"deployment_environment\": \"production\", \"detected_level\": \"unknown\", \"service_name\": \"itha\" } }",
}
```

Figure 1: Malware execution command captured by one of our honeypots

Our malware classifier flagged the sample as unusual compared with previously observed IoT malware, which prompted deeper analysis. The most notable difference was the inclusion of multiple Linux local privilege escalation exploits, a capability we do not commonly observe in IoT malware samples.

While the sample appears to be plain C, the ELF .comment section contains references to clang version 20.1.2, LLD 20.1.2, and zig-bootstrap. The most likely explanation is that it was compiled with the clang/LLD toolchain bundled with Zig. This setup makes static musl cross-compilation straightforward, allowing one host to build payloads for multiple architectures without maintaining separate GCC cross-toolchains.

We track this family as KATARU based on the ChaCha20 nonce used to decrypt the malware's configuration.

Privilege Escalation

Upon execution, the sample attempts several privilege escalation paths. Some rely on simple misconfigurations, while others appear to reuse public exploit code with no adaptation for the target architecture. Specifically:

1. It checks for severely misconfigured systems where /etc/passwd is writable by the current user. If so, it replaces root:x: with root::, removing the root password placeholder and allowing privilege escalation via su without a password to gain root.
2. If the misconfiguration is not present, the sample includes exploit code for the CVE-2026-46300 (Fragnesia) vulnerability, which allows unprivileged users to gain root access on affected Linux systems. The decompilation suggests that a publicly available exploit was copied directly with no modifications.

3. If the previous exploit fails, the sample embeds an additional exploit for the CVE-2026-43284 (DirtyFrag) vulnerability, again copied from public code.
4. The next attempted exploit is CVE-2026-31431 (Copy Fail). As with the previous exploit paths, the code appears to have been copied with little or no modification.
5. The next function prepares a cgroup v1 release_agent escape by writing a shell script that relaunched the current executable and configuring /sys/fs/cgroup/memory/release_agent to execute that script.
6. There is an additional helper function which checks if any of the files in a hardcoded list are SUID, however the output is discarded and not hooked into the rest of the privilege escalation functionality at the time.

```
118 v2 = v97;
119 if ( v97 <= 4095 )
120 {
121     fprintf((int)&unk_91B20, "target is too small: size=%lld need>=%d\n", v97, 4096);
122     goto LABEL_136;
123 }
124 strcpy(byte_BFC80, "/usr/bin/su");
125 v3 = sys_prctl(4, 1);
126 if ( v3 <= -1 )
127     die("prctl PR_SET_DUMPABLE");
128 v4 = sys_getuid32(v3);
129 v5 = sys_getgid32(v4);
130 if ( sys_pipe(&v95) <= -1 )
131     die("pipe ready");
132 v89 = v1;
133 if ( sys_pipe(v94) <= -1 )
134     die("pipe mapped");
135 v6 = fork();
136 if ( v6 <= -1 )
137     die("fork usersn mapper");
138 v7 = v6;
139 if ( v6 )
140 {
141     close();
142     close();
143     sub_69EF8(v95);
144     snprintf((int)v98, 128, "0 %u 1\n", v4);
```

Figure 2: Decompilation of exploit code found in the sample

```

590     static void enter_mapped_userns(void)
591     {
592         uid_t outer_uid = getuid();
593         gid_t outer_gid = getgid();
594         int ready_pipe[2], mapped_pipe[2], status;
595         char map[128];
596         pid_t child;
597         |
598         if (pipe(ready_pipe) < 0)
599             die("pipe ready");
600         if (pipe(mapped_pipe) < 0)
601             die("pipe mapped");
602
603         child = fork();
604         if (child < 0)
605             die("fork userns mapper");
606
607         if (child > 0) {
608             close(ready_pipe[1]);
609             close(mapped_pipe[0]);
610
611             sync_read_byte(ready_pipe[0]);
612
613             snprintf(map, sizeof(map), "0 %u 1\n", outer_uid);
614             parent_map_write_or_exit(child, "uid_map", map);
615             parent_map_write_or_exit(child, "setgroups", "deny\n");
616             snprintf(map, sizeof(map), "0 %u 1\n", outer_gid);
617             parent_map_write_or_exit(child, "gid_map", map);

```

Figure 3: Publicly available exploit code for CVE-2026-46300

The most notable aspect of these local privilege escalation attempts is the architecture mismatch. Although the analyzed sample is built for ARM, the shellcode used in the exploit code targets x86. This suggests the exploit code was copied from public sources and included verbatim, without proper testing or porting across architectures.

Persistence mechanisms

KATARU first identifies its own path with `readlink` on `/proc/self/exe`, then copies itself into several candidate locations. Some writes depend on runtime privileges; when

CAP_LINUX_IMMUTABLE is available, the malware temporarily makes the target writable, overwrites it with its own contents, and marks the resulting copy immutable and append-only.

Figure 4: List of paths to which the sample attempts to copy itself

The sample then attempts a broad persistence sweep across common Linux and embedded startup paths. The covered mechanisms include `systemd` services, timers, path units, and generators; `cron` and `@reboot`; `rc.local`, SysV init, and `inittab`; shell and profile startup files; NVRAM and U-Boot hooks; `udev`, APT, OPKG, NetworkManager, and DHCP hooks; OpenWrt `procd` and `hotplug` scripts; alternative init systems such as `runit`, `s6`, `dinit`, and `OpenRC`; XDG desktop autostart; and a watchdog loop that relaunches the executable if it exits.

Figure 5: `systemd` service created for persistence

The sample also includes Android-specific persistence. It checks several boot-script locations used by Android and root frameworks and attempts to register an Android `init` service triggered by `sys.boot_completed` or `dev.bootcomplete`.

Taken together, this is a noisy “try everything available” framework rather than a selective persistence strategy. The breadth increases the chance of surviving reboots across heterogeneous Linux, embedded, router, desktop, and Android environments, but it also reinforces the impression that the capability was assembled broadly and not validated on all of the targets.

C2 Communications

KATARU uses an encrypted custom C2 channel rather than plain-text Mirai-style communication. Each bot generates an ephemeral X25519 key pair, exchanges public keys with the server, and derives a shared secret through Diffie-Hellman. The server then proves possession of a private key corresponding to a pinned public key embedded in the binary. After the handshake, traffic is sent as length-prefixed frames that are encrypted and authenticated with ChaCha20-Poly1305 with a message counter.

This design is meant to provide forward secrecy, detect tampering, and make sink-holing harder because a third party would need the operator’s private key to impersonate the C2 server. One unusual and unlikely detail is that the pinned key matches Alice’s public key from the RFC 7748 X25519 test vectors. This suggests that the implementation may have been copied from reference material or written with LLM-assistance and integrated without any validation.

Figure 6: Pinned public key in the sample

Figure 7: RFC-7748 public key test vector containing the same key

The following commands are supported from C2:

- **Attack:** Supports multiple DDoS attack modes commonly seen in Mirai-derived botnets, including TCP, UDP, ICMP, HTTP, QUIC, and DNS floods. It also includes attack types tailored to specific services, such as Minecraft, FiveM, OpenVPN, and WireGuard. In addition, the malware implements SSH brute forcing using an embedded credential list.
- **Speed test:** Sends UDP datagrams to a target IP for a specified duration, calculates throughput based on the number of bytes sent, and reports the result back to the C2 server.

Figure 8: Disassembly of speed test function

Download and execute binary: Acts as a second-stage loader by using wget to download a binary from a C2-specified URL and execute it. The command attempts to download ELF binaries for multiple architectures using a hardcoded architecture list.

Figure 9: List of hardcoded architectures

- **Execute shell command:** Runs a command received from the C2 server via execl. The command output is not returned to the C2 server.
- **Stop attacks:** Terminates processes currently executing attack tasks.
- **Uninstall:** Stops running attacks, removes persistence artifacts and malware copies, clears related NVRAM variables, and saves the changes to persistent storage.

Figure 10: Decompilation of the uninstall function

Anti-Analysis techniques

The sample uses several anti-analysis techniques like those we have recently observed in other IoT malware. These checks may disrupt some automated sandboxes, but they are straightforward to bypass as they are all invoked from specific locations rather than being scattered throughout the binary.

Figure 11: Anti-debug functions

- **ptrace:** The sample calls ptrace with PTRACE_TRACEME to detect tracing. It also checks /proc/self/status for TracerPid and exits if the value is not 0, indicating that another process is tracing it.
- **LD_PRELOAD:** Additionally, it exits early if it detects the presence of any of the following environment variables which may indicate an analyst-controlled/instrumented execution environment: LD_PRELOAD/LD_AUDIT/LD_PROFILE/LD_DEBUG.

- **Valgrind and radare2:** Another function checks for radare2 or Valgrind-related indicators by looking for environment variables such as `R2_DEBUG`, `R2PIPE_IN`, `R2PIPE_OUT`, `VALGRIND_STARTUP_PWD`, and `RUNNING_ON_VALGRIND`. While `RUNNING_ON_VALGRIND` exists as a Valgrind macro, we are not aware of a corresponding environment variable with this name, so this appears to be a mistaken check.
- **Process names:** The next check looks for process names associated with debuggers or common reverse engineering tools. If one is found, the sample terminates.

Figure 12: Process names associated with reverse engineering tools

- **Timing check:** the sample computes the time delta between the execution of a short loop and terminates if it is larger than expected (~100ms), which indicates single-stepping or heavy instrumentation.
- **Code integrity:** The final check hashes the process image's `.text` section and compares the result with the 32-byte digest stored in the ELF `.bothash` section. In this build, the check is effectively inert because the `.bothash` digest is set to zero bytes, bypassing the comparison. The intended purpose is to detect breakpoints or code patches.

Decoy Layer

One of KATARU's more unusual features is a detached thread that periodically emits fake C2-like traffic. This may be intended to confuse sandbox analysis and automated IOC extraction, but it could also be a development or testing artifact used to exercise multiple protocol handlers.

The decoy traffic includes HTTP-style requests such as `POST /api/v1/beacon` and `GET /api/v1/query`, fake headers including `X-Bot-Id` and `X-Campaign: op-cascade-2024`, and a JSON body containing a hardcoded decoy JWT. These requests rotate across fake hostnames, several of which are crafted to resemble update services, telemetry relays, management panels, or C2 infrastructure.

Figure 13: JWT token containing "decoy" in its name

KATARU also generates fake IRC bot chatter, attempts TCP connections to port 443 on decoy IPs, and sends junk UDP bursts to DNS-adjacent ports. The decoy IP list deliberately mixes RFC documentation ranges with real third-party infrastructure such as GitHub and Cloudflare, increasing the risk that automated reports will incorrectly include benign infrastructure as indicators of compromise.

Figure 14: TEST-NET non-routable IP ranges used by UDP sweep function

The binary also plants unused strings in `.rodata`, including Mirai/Katana-style banners, CNC labels, scanner controls, attack names, and a classic Mirai loader command. These strings are not referenced by code, so relying on raw `strings` output could misclassify the sample as a stock Mirai or Katana variant with autonomous Telnet scanning. In this build, spreading is not implemented; the SSH credential attack is C2-tasked rather than self-propagating.

Development Artifacts and Reuse Signals

Several implementation artifacts strongly suggest that KATARU was assembled with AI assistance rather than developed and validated as a mature malware family. The malware includes an unusually large number of persistence mechanisms spanning Linux, embedded Linux, router, desktop, and Android environments, which would be difficult to develop and reliably test manually across all supported targets. Its LPE chain also shows clear portability issues, including x86 shellcode embedded in the analyzed ARM sample, suggesting blind copy-pasting from public sources. Other indicators point to generated or stitched-together implementation patterns, including decoy infrastructure values that use RFC documentation IP ranges and a pinned X25519 public key matching a known test vector. Taken together, these artifacts suggest an operator used public examples, exploit PoCs, and LLM-generated assistance to rapidly combine capabilities without fully reviewing or validating the result.

Mitigations and Recommendations

Figure 15: Detection of KATARU malware transfer and C2 connection attempt

Figure 16: List of assets interacting via the Telnet protocol

- **Reduce exposed management access:** Disable Telnet where possible, replace default or weak credentials, and restrict device administration to trusted networks or VPN-accessible management planes.
- **Detect network anomalies around IoT and OT assets:** Use network visibility and anomaly detection to identify behavior that deviates from each device's normal baseline, including unusual outbound connections, unexpected protocol usage, beacon-like traffic, scanning behavior, sudden traffic-volume spikes, or DDoS-like patterns. This is especially important for embedded devices where host-level telemetry is limited or unavailable. To see how the Nozomi Networks Platform applies these capabilities in practice, [request a demo today](#).

- **Enforce segmentation for exposed IoT and OT devices:** Place embedded devices in dedicated network zones, restrict east-west communication, and allow only required management and operational traffic between segments. Segmentation limits lateral movement, reduces the blast radius of compromised devices, and makes abnormal cross-zone communication easier to detect and investigate.
- **Patch firmware and vulnerable kernels:** Keep device firmware and Linux kernels current where vendor support allows. The embedded privilege escalation attempts highlight why outdated kernels, and unmaintained embedded distributions remain high-risk targets.
- **Monitor persistence artifacts:** Watch for unexpected changes to startup locations such as systemd units, cron entries, rc scripts, OpenWrt hotplug paths, Android boot scripts, package-manager hooks, and files marked immutable or append-only.
- **Investigate unusual outbound activity:** Prioritize alerts involving unfamiliar TCP connections, encrypted custom protocols, traffic to known infrastructure, and sudden increases in flood-like outbound traffic from embedded assets.

Conclusion

KATARU illustrates how commodity IoT malware can absorb newer capabilities without necessarily becoming mature or well-engineered. The sample retains familiar Mirai-style botnet behavior but expands it with multiple Linux privilege-escalation attempts, broad persistence across Linux and embedded environments, encrypted custom C2, anti-analysis checks, and a decoy layer that can mislead automated analysis. At the same time, architecture-mismatched exploit code, RFC-derived test values, unused Mirai-themed strings, and unusually broad cross-platform logic strongly point to rapid integration with AI assistance and limited validation. For defenders, the key takeaway is that familiar IoT malware foundations can quickly incorporate public exploit code and new operational features. Reducing exposed management services, monitoring persistence artifacts, and investigating unusual outbound behavior from embedded assets remain critical for identifying and containing similar threats.

MITRE ATT&CK Mapping

- **Initial Access**
 - T1110.001** – Brute Force (Password Guessing): Initial access was observed through Telnet credential brute forcing, and the sample also includes brute-force capability for remote services.

- **Execution**
 - **T1059.004** – Unix Shell: Shell staging (wget) and execution of downloaded binaries.
 - **T1106** – Native API: Raw syscall wrapper, raw sockets, and ioctl for file-attribute changes.
- **Persistence**
 - **T1543.002** – systemd Service: Installs a network-dispatch service, timer, path unit, and generator.
 - **T1037.004** – RC Scripts: Falls back across init.d, rc.local, inittab, procd, OpenRC, s6, and others.
 - **T1053.003** – Cron: Adds /etc/cron.d and crontab entries.
- **Privilege Escalation**
 - **T1068** – Exploitation for Privilege Escalation: Multi-stage LPE chain.
- **Defense Evasion**
 - **T1027** – Obfuscated Files or Information: XOR-0x22 config layer plus a ChaCha20 string vault (nonce "KATARU").
 - **T1562.001** – Impair Defenses: Disables the hardware watchdog, sysrq, and reboot.
- **Discovery**
 - **T1082** – System Information Discovery: Reads uname, /proc/cpuinfo, disk, and information about page count and page sizes.
 - **T1016** – System Network Configuration Discovery: Reads /proc/net/*, resolv.conf, and the local IP.
- **Command and Control**
 - **T1573.002** – Encrypted Channel: X25519 key exchange with ChaCha20-Poly1305 AEAD frames.
 - **T1095** – Non-Application Layer Protocol: Custom length-prefixed binary protocol over TCP (port 6767).
 - **T1105** – Ingress Tool Transfer: Fetches per-architecture payloads via bins.sh.
- **Impact**
 - **T1498.001** – Direct Network Flood: UDP, TCP SYN/ACK, GRE, and ICMP floods.
 - **T1499.002** – Service Exhaustion Flood: HTTP, HTTP/2, QUIC, and game-server floods.

IoCs

- cc76bc218627279ecb4d0ce74ad2651e9db9e3e843e35d6569576e056e3a9218 (loader)
- 13382c16e2401b07451577b46e634b8031ec254d98b876e59692b5fa22abc1d4 (arm32)
- 6fbae3505ae0d638b820165c572d548ce92dda71e82dc47e8efe13f30617f35f (arm32)
- 9d87e6615c810907443ebd5e915f3b35099c3b5c6b6c684637138a7f8ec9cebc (arm32)
- 9d7cd4948a1fcbaeadc425752fce9a933bd6fc41eeede030dff7b99b3bc51d5 (amd64)

- 160[.]191.242.92 (Telnet credential brute force and C2)

YARA

```
rule IOT_DDOS_KATARU_sample
{
    meta:

        name = "KATARU - DDOS"

        description = "KATARU modern Mirai variant"

        hash1 =
"9d87e6615c810907443ebd5e915f3b35099c3b5c6b6c684637138a7f8ec9cebc"

        hash2 =
"6fbae3505ae0d638b820165c572d548ce92dda71e82dc47e8efe13f30617f35f"

        hash3 =
"13382c16e2401b07451577b46e634b8031ec254d98b876e59692b5fa22abc1d4"

    strings:

        $kataru = "KATARU\x00"

        $bothash = ".bothash"

        $network_dispatch = "network-dispatch"

        $kwork_0 = "[jbd2/dm0-8]"

        $kwork_1 = "[kworker/u:2]"

        $kwork_2 = "[kworker/u:1]"

        $kwork_3 = "[kworker/2:1]"

        $kwork_4 = "[kworker/1:1]"

        $kwork_5 = "[kworker/0:1]"

        $kwork_6 = "[kblockd/1]"
```

```
$kwork_7 = "[kworker/u:0]"
$kwork_8 = "[kworker/3:0]"
$kwork_9 = "[kworker/2:0]"
$kwork_10 = "[kworker/1:0]"
$kwork_11 = "[kworker/0:0]"
$kwork_12 = "[kthrotld/0]"
$kwork_13 = "[kblockd/0]"
$str_0 = "TracerPid:"
$str_1 = "/proc/self/status"
$str_2 = "/tmp/.cg_escape.sh"
$str_3 = "/sys/fs/cgroup/memory/release_agent"
$str_4 = "[getent passwd root] %s"
$str_5 = "id; whoami; cat /etc/shadow | head -2; exit"
$cnc_0 = "POST /api/v1/beacon"
$cnc_1 = "X-Bot-Id"
$cnc_2 = "X-Campaign"
$cnc_3 = "X-Arch"
$cnc_4 = "NICK bot-%s"
$cnc_5 = "USER %s 8 * :%s"
$cnc_6 = "JOIN #%s"
$cnc_7 = "PRIVMSG #%s :HTTPFLOOD ready"
$cnc_8 = "eyJhbGciOiJIUzI1NiJ9.decoy.token"
$cnc_9 = "
{"id\":"%s\","cmp\":"%s\","stage\":"beacon\","uptime\":"%u\","token\":"
```

condition:

uint32(0) == 0x464c457f and

filesize < 1MB and

(

(

any of (\$bothash, \$network_dispatch, \$kataru)

and

(

2 of (\$str_*) and

2 of (\$cnc_*)

)

) or

(

#network_dispatch > 20 and

(

6 of (\$kwork_*) or

2 of (\$str_*) or

2 of (\$cnc_*)

)

)

)

}

References:

- <https://github.com/v12-security/pocs/tree/main/fragnesia>

- <https://github.com/v4bel/dirtyfrag>
- <https://xint.io/blog/copy-fail-linux-distributions>
- <https://www.rfc-editor.org/info/rfc7748/>
- <https://www.rfc-editor.org/info/rfc5737/>



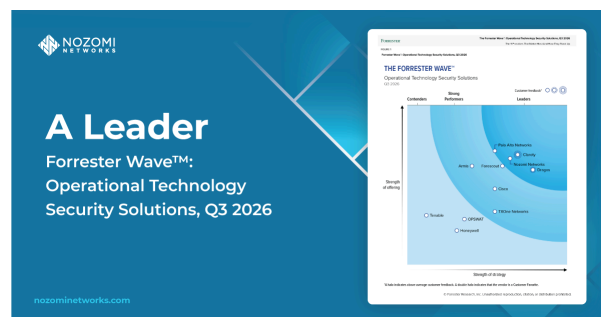
Zero-Days at AI Speed: A Wake-Up Call for Critical Infrastructure

AI has collapsed the cost of finding vulnerabilities, turning what once took expert days into automated pipelines that work in minutes. But response hasn't kept pace. Disclosure cycles, patch testing, and operational constraints haven't changed. Here's what that widening gap means for OT security, and what it actually takes to close it.

[View Blog](#)

Tengu: A Modernized Mirai That Doesn't Want to Leave

[View Blog](#)



The Forrester Wave™: Operational Technology Security Solutions, Q3 2026

Read the The Forrester Wave™: Operational Technology Security Solutions to see why Nozomi Networks was named a Leader.

[View Resource](#)