GOst in the Protocol: Hunting Ligolo with JARM Fingerprinting in the wild

May 17, 2025

Super TL;DR: You can git clone Ligolo and connect to Ligolo redirection proxies on the Internet. We have 3 JARM signatures to search for them, one is identical to Sliver C2 (default Go TLS is very signaturable). We created a custom ligolo agent that can verify if a server is a Ligolo proxy. We do not advise you check-in as an agent to foreign Ligolo redirection servers. They are probably APTs or threat actors.

Regular TL;DR: We identified three distinct JARM signatures that reliably identify Ligolo proxy servers in the wild: one for Ligolo 0.7.x, one for Ligolo 0.8.x, and one for Ligolo-MP (which is shared with Sliver C2). These signatures indicate at minimum a default Go TLS implementation. To definitively confirm Ligolo proxies, we developed a 4-stage verification methodology using a modified Ligolo agent that tests whether the server correctly implements the yamux protocol. For distinguishing between Ligolo-MP and Sliver C2 (which share the same JARM signature), we discovered a key difference: Sliver responds with a "bad certificate" error, while Ligolo-MP fails with "remote error: tls: certificate required." Our research demonstrates that Ligolo proxies are highly vulnerable to fingerprinting, making them dangerous to expose on the Internet. Finally, do not attempt to connect to public potential Ligolo proxies with authorization.

This blog post is for educational and research purposes.

The Hunt

If you want the primer, read port9's blog on hunting Sliver C2 in the wild, largely using JARM fingerprinting as well.

This post was inspired by and builds on port9's work and asks:

1. Can we identify Ligolo proxy servers on the internet?
2. If yes, can you just connect to them with a Ligolo agent? The answer to both is yes.

What is JARM?

JARM is an active TLS fingerprinting tool developed by Salesforce that sends 10 specially crafted TLS Client Hello packets to a server and analyzes the responses. These responses are combined into a 62-character fingerprint that can identify specific applications based on their underlying TLS implementation.

For those in the back: JARM fingerprinting ignores x509 certificates, it only cares about the UNDERLYING TLS implementation.

JARM also does not care about what service is advertised on the port.

Ligolo's JARM signatures

Our analysis revealed three distinct JARM signatures associated with Ligolo (we git cloned Ligolo locally and ran them):

Ligolo-ng releases Ligolo-MP releases

1. **Ligolo 0.7.x**: `40d1db40d00040d1dc43d1db1db43d5ecfbe778b06e32b538bd51f24eb7398`
2. **Ligolo 0.8.x**: `40d40d40d00040d00043d40d40d43d70e44c2d581076ca8e0c7ff40bb556f2`
3. **Ligolo-MP 2.0.x**: `00000000000000000043d43d00043de2a97eabb398317329f027c66e4c1b01` (this is also Sliver C2's signature)

Ligolo vs Production Systems

Service	JARM Signature
Google	`27d40d40d29d40d1dc42d43d00041d4689ee210389f4f6b4b5b1b93f92252d`
Cloudflare	`27d40d40d00040d1dc42d43d00041d6183ff1bfae51ebd88d70384363d525c`
x.com	`29d29d15d29d29d00042d42d0000005fd00fabd213a5ac89229012f70afd5c`
proton.me	`29d29d15d29d29d00042d42d0000005fd00fabd213a5ac89229012f70afd5c`
Ligolo 0.7.x	`40d1db40d00040d1dc43d1db1db43d5ecfbe778b06e32b538bd51f24eb7398`
Ligolo 0.8.x	`40d40d40d00040d00043d40d40d43d70e44c2d581076ca8e0c7ff40bb556f2`
Ligolo-MP and Sliver C2	`00000000000000000043d43d00043de2a97eabb398317329f027c66e4c1b01`

Notice how production systems have rich diversity in response bytes and non-trivial hashes suggesting deeper feature support, while Ligolo signatures (especially Ligolo-MP) show patterns that immediately stand out as anomalous.

Anomalous and "this is definitely ligolo" are not the same thing, immediate case in point is Sliver C2 has the same JARM as Ligolo-MP.

Sliver C2 and Ligolo-MP JARM Collision

Our most surprising discovery was that Ligolo-MP 2.0.x shares its exact JARM signature with Sliver C2's mutual TLS (mTLS) configuration:

```
Shodan query:
ssl.jarm:"00000000000000000043d43d00043de2a97eabb398317329f027c66e4c1b01"
```

Search for Ligolo-MP and Sliver C2 using Shodan: https://www.shodan.io/search?query=ssl.jarm%3A%2200000000000000000043d43d00043de2a97eabb398317329f027c66e4c1b01%22

This signature collision tells us both tools use the same TLS libraries/configurations, which is not surprising given that both are modern mTLS Golang tools.

Back in 2022, Sliver's author Moloch (@LittleJoeTables) stated:

> *This is gonna be a "won't fix" from us, so will be a reliable signal for threat hunting. If you expose your control interfaces you deserve to get a caught :-)*

Lol, so BOGO special going on for threat hunters.

This leave us with 2 questions:

1. How can tell Ligolo from some random Go TLS thing?
2. How can identify Ligolo-MP from Sliver C2?

How to confirm Ligolo 0.7.x/0.8.x

JARM signatures cut down the Internet to a more manageable set of candidates, but they don't provide definitive confirmation.

To solve this problem we created a modified Ligolo agent that could connect to potential Ligolo proxies and verify if they're genuine. (As we will continue to implore, **do not** do this live on the Internet without explicit authorization.)

The key modifications were made to the `main.go`, where we added detailed logging of the yamux protocol messages.

Rules with our custom Ligolo Agent:

1. If yamux is not implemented, then it can't be Ligolo.
2. Any immediate TLS, HTTP, or other protocol errors are not Ligolo.
3. If yamux works and we start seeing `YAMUX READ` messages coming from the server, then we just checked in as an agent and it's confirmed Ligolo (yes seriously.)

Breakdown of what we added:

1. **Yamux Stream Establishment**: The agent attempts to open a yamux stream with the target server. Only a genuine Ligolo proxy would properly handle this request, as yamux is the multiplexer protocol used by Ligolo for tunneling.
2. **Protocol Behavior Verification**: The agent tests Ligolo-specific protocol behavior by sending control packets with specific command bytes. A genuine Ligolo proxy would process these packets correctly.
3. **Multiple Stream Support**: The agent verifies that the server can handle multiple simultaneous streams, which is a core feature of Ligolo's multiplexing capability.

4. **Connection Stability**: Finally, the agent tests if the connection remains stable after a delay, as genuine Ligolo proxies maintain connections until explicitly terminated.

Custom code that we added

Here’s how we implemented the verification process in code:

```go
// LoggingConn is a wrapper around net.Conn that logs all yamux protocol messages
type LoggingConn struct {
    net.Conn
    readCount  int
    writeCount int
}

// Read intercepts and logs all data read from the connection
func (l *LoggingConn) Read(b []byte) (n int, err error) {
    n, err = l.Conn.Read(b)
    if n > 0 {
        l.readCount++
        log.Infof("[YAMUX READ #%d] %d bytes:", l.readCount, n)
        // Parse and display yamux header if present
        if n >= 12 {
            parseYamuxHeader(b[:12])
        }
        // Display hex dump of data
        fmt.Println(hex.Dump(b[:n]))
    }
    return
}

// Write intercepts and logs all data written to the connection
func (l *LoggingConn) Write(b []byte) (n int, err error) {
    n, err = l.Conn.Write(b)
    if n > 0 {
        l.writeCount++
        log.Infof("[YAMUX WRITE #%d] %d bytes:", l.writeCount, n)
        // Parse and display yamux header if present
        if n >= 12 {
            parseYamuxHeader(b[:12])
        }
        // Display hex dump of data
        fmt.Println(hex.Dump(b[:n]))
    }
    return
}

// Modified connect function with verification capabilities
func connect(conn net.Conn) (int, error) {
    // Create a logging wrapper around the connection for protocol analysis
    loggingConn := &LoggingConn{Conn: conn}

    // Initialize yamux server with the logging connection
    yamuxConn, err := yamux.Server(loggingConn, yamux.DefaultConfig())
    if err != nil {
        log.WithError(err).Error("Failed to create yamux server")
        return 0, err // Stage 1 failed
    }

    log.Info("STAGE 1: Attempting to open a yamux stream")
```

```
    // Stage 1: Open a stream
    stream, err := yamuxConn.Open()
    if err != nil {
        log.WithError(err).Error("Failed to open yamux stream")
        return 0, err // Stage 1 failed
    }
    log.Info("STAGE 1 PASSED: Successfully opened yamux stream")

    log.Info("STAGE 2: Testing Ligolo-specific protocol behavior")
    // Stage 2: Send control packet (command byte 0x01)
    _, err = stream.Write([]byte{0x01, 0x12, 0x34, 0x56, 0x78})
    if err != nil {
        log.WithError(err).Error("Failed to write control packet")
        return 1, err // Stage 2 failed
    }
    log.Info("STAGE 2 PASSED: Successfully sent control packet")

    // Stage 3: Multiple streams test
    log.Info("STAGE 3: Testing multiple stream support")
    stream2, err := yamuxConn.Open()
    if err != nil {
        log.WithError(err).Warn("Failed to open second stream - This may be a
partial implementation")
        return 2, err // Stage 3 failed
    }
    log.Info("STAGE 3: Successfully opened second stream")

    // Write test data to second stream
    _, err = stream2.Write([]byte{0x02, 0x00})
    if err != nil {
        log.WithError(err).Warn("Failed to write to second stream - This may be a
partial implementation")
        return 2, err // Stage 3 failed
    }
    log.Info("STAGE 3 PASSED: Successfully wrote to second stream")

    // Stage 4: Stability test
    log.Info("STAGE 4: Testing connection stability after delay")
    time.Sleep(2 * time.Second) // Wait to test connection stability

    // Try writing to stream after delay
    _, err = stream.Write([]byte{0x03, 0x00})
    if err != nil {
        log.WithError(err).Warn("STAGE 4 FAILED: Unable to write after delay: %s -
Connection unstable", err)
        return 3, err // Stage 4 failed
    }

    log.Info("STAGE 4 PASSED: Connection remains stable after delay")
    log.Info("ALL VERIFICATION STAGES PASSED")
    return 4, nil // All stages passed
}
```

The money is in seeing any `YAMUX READ` in the output. If you see it, then you are communicating with a Ligolo proxy via the yamux protocol (and your agent checked in successfully):

```
./agent-enhanced -connect 192.168.1.115:8443 --ignore-cert
...
[YAMUX READ #5] 12 bytes:
=== YAMUX HEADER ===
Version:   0
Type:      PING (0x02)
Flags:     0x0002
Stream ID: 0
Length:    0 bytes
====================
00000000  00 02 00 02 00 00 00 00  00 00 00 00              |............|
```

The custom Ligolo agent provides detailed output for each stage, making it easy to identify exactly where a potential Ligolo proxy fails the verification process.

Again, this methodology allows us to definitively distinguish between:

- Genuine Ligolo proxies (pass all four stages + `YAMUX READ` messages)
- Partial implementations, honeypots, messed up deployments, custom deployments, etc. (pass some stages but fail others)
- Completely unrelated services (fail at certain stages, show invalid protocols, invalid HTTP responses, completly unrelated errors, etc.)

How to distinguish Ligolo-MP from Sliver C2

We have the JARM signature, but how do we tell the difference between Ligolo-MP and Sliver C2?

1. Running a Ligolo agent against a Ligolo-MP server will fail because Ligolo-MP requires a client certificate to connect.
2. How do we tell the difference between Ligolo-MP and Sliver C2?

We made a pretty simple discovery: key differences in how they handle certificate validation.

The following is how they react when you try to connect to each with our custom Ligolo agent:

Download our custom Ligolo agent from here

1. **Ligolo-MP**:
    - Immediately fails with “remote error: tls: certificate required” when connecting without a client certificate
    - Connection is terminated at the TLS handshake phase
    - Our verification agent reports this as “PARTIALLY VERIFIED” because the initial connection works but fails before yamux protocol verification

```
./agent-enhanced -connect 192.168.1.115:8443 --ignore-cert
...
[ERR] yamux: Failed to read header: remote error: tls: certificate required <---
Ligolo-MP
WARN[0002] STAGE 4 FAILED: Unable to write after delay: stream closed - Connection
unstable
INFO[0002] VERIFICATION RESULT: PARTIALLY VERIFIED - This appears to be a Ligolo proxy
but may be a partial implementation or honeypot
ERROR[0002] Connection error: remote error: tls: certificate required <--- Ligolo-MP
FATAL[0002] remote error: tls: certificate required <--- Ligolo-MP
```

2. **Sliver C2**:
    - Allows initial connection and yamux protocol communication is attempted
    - Passes stages 1-3 of our verification process
    - Fails at stage 4 with “bad certificate” error after successful yamux protocol interaction
    - Our verification agent reports this as “PARTIALLY VERIFIED” but with a different error pattern, which makes perfect sense

```
./agent-enhanced -connect 192.168.1.115:31337 --ignore-cert
...
INFO[0000] STAGE 3 PASSED: Successfully wrote to second stream
INFO[0000] STAGE 4: Testing connection stability after delay
[ERR] yamux: Failed to read header: remote error: tls: bad certificate
WARN[0002] STAGE 4 FAILED: Unable to write after delay: stream closed - Connection
unstable
INFO[0002] VERIFICATION RESULT: PARTIALLY VERIFIED - This appears to be a Ligolo proxy
but may be a partial implementation or honeypot
ERROR[0002] Connection error: remote error: tls: bad certificate <--- Sliver C2
FATAL[0002] remote error: tls: bad certificate <--- Sliver C2
```

Ligolo-MP = `remote error: tls: certificate required` Sliver C2 = `remote error: tls: bad certificate`

This difference in certificate validation provides a reliable method to distinguish between these two frameworks after we’ve narrowed down candidates using JARM fingerprinting.

Do not connect to an APT or threat actor’s Ligolo proxy

While our custom Ligolo agent could theoretically confirm if an internet-accessible server is running Ligolo 0.7.x, 0.8.x, or Ligolo-MP 2.0.x, we advise against connecting to unknown servers without proper authorization:

Attempting to connect to servers you don’t own could be:

1. Potentially illegal under computer fraud and abuse laws
2. Dangerous to you: you would be checking in to a potential APT or threat actor Ligolo proxy.

Be careful running open source tools

This highlights a critical vulnerability of Ligolo deployments: **anyone** can grab the source code, create a custom agent and attempt to connect to internet-accessible Ligolo servers.

For pen testers/Red Teamers: Be careful exposing open source tools publicly. Even going to the trouble of gaining a legitimate certiticate, the JARM will still identify you. You are being mixed together with APTs, malicious actors, other Red Teams, random people etc. As a Red Team, a malicious APT can connect to your Ligolo 0.7.x or 0.8.x server. Red Teams should think about the potential ramifications of this.

Why This Matters

Our findings have significant implications for security professionals:

1. **For Blue Teamers**: JARM fingerprinting is already effective. Keep adding JARM fingerprints to your detection lists. They're not perfect, but they are still useful.
2. **For Red Teamers**: We highly recommend you do not expose your ligolo proxy to the Internet.
3. **For Researchers**: We're sure you understand how dangerous this is. This could lead to some interesting scenarios.
4. **For APT Hunters**: When searching for those JARM signatures on Internet mapping services like Shodan or Censys, take a look at the other (potentially false) services running, ports, etc. which can raise red flags. However, do not think that because only 443 is open and shows a legitimate cert that it is safe to ignore as "not Ligolo". As we've shown, **the JARM doesn't confirm, but it doesn't lie!**