EXPATCH LLC

VULNERABILITY RESEARCH · COORDINATED DISCLOSURE

We Trusted Telegram With Our Research to Protect Users — and Were Written Out of the Story

A technical account of NTLM relay, heap-memory disclosure, and the remediation history across Telegram Desktop and QtNetwork.

Alexander Rostilov
Denis Rostilov
ExPatch Vulnerability Research Team

September 2026
Telegram Desktop 6.8.x · Qt 6.11.1
Boston, Massachusetts

EXECUTIVE SUMMARY

What Happened

ExPatch research identified two distinct security issues in a QtNetwork authentication path reachable from the official Telegram Desktop client: NTLM authentication relay and repeatable disclosure of process memory.

Automatic NTLM

A 407 response from a controlled proxy could start SSPI and generate an NTLM response for the current Windows session without separate user consent.

Laboratory RCE Chain

In an isolated domain, relay to AD CS led to code execution within the compromised account's privileges. We obtained SYSTEM privileges when the victim account was a Domain Admin.

Up to 65 KB per Exchange

A flaw in qExtractServerTime() included up to 64,992 bytes of uninitialized heap memory in the NTLMv2 response; 100 cycles disclosed approximately 6.5 MB.

Attribution

The full report reached Telegram on June 6; matching patches are dated June 8; Qt upstream work began on June 10, and the exact heap fix followed on June 12. The researchers are not named in the public records reviewed.

PRIMARY ANALYSIS	LIBRARY	RCE ENVIRONMENT
Telegram Desktop 6.8.4	**Qt 6.11.1 / QtNetwork**	**Telegram Desktop 6.8.2**

NAVIGATION

Contents

> "Our experience is shaped by our mission to protect our users in authoritarian regimes."

— Pavel Durov, September 5, 2024; original post.

Fig. 1. *Editorial excerpt from Pavel Durov's original post: the author header, quoted paragraph, and publication metadata are preserved; omitted passages are marked with ellipses.*

We agree with these words. More than that—we believed them.

For someone living under censorship, a Telegram proxy is more than a setting in an application. At times, it is the last narrow path to independent information and the open internet. But that path to freedom becomes a trap if it can silently disclose credentials, expose process memory, or give an attacker control of the device.

After discovering exactly such a risk, we could have reported it to Qt first, established a record of our discovery, and waited through the usual remediation cycle. We chose differently. The night before disclosure, we did not sleep: we revalidated the attack chain, assembled the technical evidence, and documented every stage so Telegram engineers could reproduce the issue immediately. On June 6, we sent Telegram a complete technical report, explicitly marked confidential and prohibited from distribution without permission from ExPatch LLC. Telegram controlled the application and could block the most dangerous path faster than anyone else. We therefore put user safety ahead of sleep, convenience, and the opportunity to establish priority with Qt first.

Telegram was given the first opportunity to protect its users—and used it. Soon afterward, corresponding fixes appeared in Qt, while we were neither informed of upstream coordination nor included in it, and neither the researchers nor ExPatch appeared in the public attribution. Was this routine upstream coordination, or a breach of the trust that gave Telegram its head start? This article presents the documents, code, and timeline so readers can decide for themselves.

The research was conducted against Telegram Desktop 6.8.4, which used Qt 6.11.1.

From Proxy Handling to NTLM

The investigation began with the architecture of Telegram Desktop's proxy subsystem. We established that HTTP and SOCKS5 proxy connections are delegated to Qt. At this stage, our focus was not a particular library or company, but the trust boundary itself: what happens when a hostile proxy server controls responses during connection establishment? That question became the starting point for the audit.

For almost a month, we examined this mechanism from multiple angles: trust boundaries, state transitions, and ways to force the client into unexpected behavior. Most directions eventually reached a dead end, and the attack surface appeared nearly exhausted. The turning point came when we found that, during an active proxy exchange, Qt accepted a demand to switch authentication to NTLM. A modern proxy connection could silently transition to a legacy mechanism associated with NTLM Relay attacks for decades.

At a high level, the chain is remarkably simple. Telegram Desktop connects through a configured HTTP proxy and Qt sends a CONNECT request. A hostile proxy replies with `407 Proxy Authentication Required` and advertises `Proxy-Authenticate: NTLM`. Qt treats the response as a normal authentication challenge, starts NTLM automatically, and sends the first handshake message—NTLM Type 1 (Negotiate). No explicit user consent is requested.

The key question was which checks Qt performed before allowing NTLM to use the user's system credentials. Because NTLM Relay has been understood for decades, we expected a control that bound authentication to the intended proxy and prevented the same exchange from being replayed to another service. The implementation contained no such protection: Qt did not validate an expected SPN, did not bind the exchange to a protected channel through EPA, and did not require exchange integrity through MIC. An attacker controlling the proxy did not need to crack NTLM or recover a password; relaying the authentication messages to another service in real time was enough.

This finding changed the investigation completely. NTLM was not hidden behind an obsolete enterprise configuration—it could be triggered in the middle of an ordinary proxy exchange. After a victim connected to the attacker-controlled HTTP proxy, the attacker needed only one normal-looking response: `407 Proxy Authentication Required` with `Proxy-Authenticate: NTLM`. A single header caused Qt to involve the user's system credentials in an NTLM exchange automatically.

How Qt Starts NTLM Without Confirmation

To understand why one proxy response was sufficient, we traced the complete execution path from processing the 407 response to generating the authentication message. The path consists of three levels.

Level 1 — Qt Accepts the 407 Response

Telegram Desktop configures the proxy through `QTcpSocket::setProxy`. In `qhttpsocketengine.cpp:562-599`, Qt processes the proxy's response to CONNECT. A `407 Proxy Authentication Required` status is passed to the built-in authentication mechanism:

qhttpsocketengine.cpp — 407 handling · lines 562–599

```
} else if (statusCode == 407) {                          // :562
    if (d->authenticator.isNull())
        d->authenticator.detach();

    priv = QAuthenticatorPrivate::getPrivate(
        d->authenticator
    );

    const auto headers = d->reply->header();

    priv->parseHttpResponse(
        headers,
        true,                                             // isProxy=true
        d->proxy.hostName()
    );                                                    // :567
}
```

NTLM has not yet been selected at this level. The important detail is that Qt passes every proxy-controlled response header to `parseHttpResponse()` with `isProxy=true`. The authentication method is therefore selected from attacker-controlled `Proxy-Authenticate` content. The next level shows how the string `NTLM` becomes an instruction to start system authentication.

Level 2 — Qt Selects the Authentication Scheme

The transition to NTLM is chosen inside `QAuthenticatorPrivate::parseHttpResponse()` in `qauthenticator.cpp:466-558`. Because the function was called with `isProxy=true`, Qt examines the proxy-controlled `Proxy-Authenticate` header:

qauthenticator.cpp — scheme selection · lines 466–558

```
const auto search = isProxy
    ? QHttpHeaders::WellKnownHeader::ProxyAuthenticate
    : QHttpHeaders::WellKnownHeader::WWWAuthenticate;

// isProxy=true  → Proxy-Authenticate
// isProxy=false → WWW-Authenticate

for (const auto &current : headers.values(search)) {
    if (method < Basic
        && str.startsWith("basic"_L1, Qt::CaseInsensitive)) {
        method = Basic;

    } else if (method < Ntlm
               && str.startsWith("ntlm"_L1,
                                 Qt::CaseInsensitive)) {
        method = Ntlm;                                  // :488-490
        headerVal = QByteArrayView(current).mid(5);

    } else if (method < DigestMd5
               && str.startsWith("digest"_L1,
                                 Qt::CaseInsensitive)) {
        // ...

    } else if (method < Negotiate
               && str.startsWith("negotiate"_L1,
                                 Qt::CaseInsensitive)) {
        // ...
    }
}

// ...

case Ntlm:
case Negotiate:
    // work is done in calculateResponse()
    break;
```

An additional problem is that `method` is reset before each new response is parsed. A proxy can therefore demand NTLM not only in the first response, but in a later 407 and change the scheme while the connection is already being established. Once `Ntlm` has been selected, the function performs no additional trust checks; generation of the system response is deferred to `calculateResponse()`.

Level 3 — Qt Starts NTLM Without Notifying the Application

After selecting the scheme, execution returns to `qhttpsocketengine.cpp:574-599`:

qhttpsocketengine.cpp — starting the NTLM phase · lines 574–599

```
if (priv->phase == QAuthenticatorPrivate::Done
    || (priv->phase == QAuthenticatorPrivate::Start
        && (priv->method == QAuthenticatorPrivate::Ntlm
            || priv->method == QAuthenticatorPrivate::Negotiate))) {

    if (priv->phase == QAuthenticatorPrivate::Start)
        priv->phase = QAuthenticatorPrivate::Phase1;     // :578-580

    if ((priv->method != QAuthenticatorPrivate::Ntlm
         && priv->method != QAuthenticatorPrivate::Negotiate)
        || credentialsWasSent) {

        proxyAuthenticationRequired(
            d->proxy,
            &d->authenticator
        );                                               // :596-597
    }
}
```

The decisive point is the condition before `proxyAuthenticationRequired()`. With an ordinary scheme, Qt emits this signal so the application can request credentials or refuse authentication. During the first transition to NTLM or Negotiate, however, `credentialsWasSent == false`, making the entire condition false. No signal is emitted.

Telegram Desktop consequently has no opportunity to approve, warn about, or block the transition. Qt moves the authenticator to `Phase1`, calls `calculateResponse()`, asks Windows SSPI for system SSO credentials, and sends an NTLM Type 1 (Negotiate) message to the proxy. The transition is silent not only to the user, but to the application itself.

Why does this permit NTLM Relay? The critical issue is not a flaw in SSPI itself, but the way Qt invokes it during automatic proxy authentication. If `QAuthenticator` contains no username, Qt does not stop the connection or ask the application for credentials. It calls Windows SSPI and obtains the default credentials of the current logon session. In a domain environment, a hostile proxy can therefore initiate an NTLM exchange on behalf of the user's corporate account without the user's knowledge.

Stage 1 — `calculateResponse()` Selects System Authentication

The first confirmation appears in `qauthenticator.cpp:585-592`. When the proxy requests NTLM and no challenge has yet been received, Qt begins the first handshake phase:

qauthenticator.cpp — calculateResponse() · lines 585–592

```
case QAuthenticatorPrivate::Ntlm:
    if (challenge.isEmpty()) {
#if QT_CONFIG(sspi)
        QByteArray phase1Token;

        if (user.isEmpty()) {
            // Only pull from system if no user was
            // specified in authenticator
            phase1Token = qSspiStartup(
                this,
                method,
                host
            );
        }

        // ...
#endif
    }

    // ...
    break;
```

The Qt comment is explicit: “Only pull from system if no user was specified in authenticator.” This is where a proxy-controlled demand for NTLM is connected to the current Windows account.

Stage 2 — System Credentials in `qSspiStartup()`

qauthenticator.cpp — qSspiStartup() · lines 1566–1607

```
SEC_WINNT_AUTH_IDENTITY auth;
auth.Flags = SEC_WINNT_AUTH_IDENTITY_UNICODE;

bool useAuth = false;

if (method == QAuthenticatorPrivate::Negotiate
    && !ctx->user.isEmpty()) {
    // Explicit credentials are used only
    // for Negotiate with a non-empty user
    useAuth = true;
}

SECURITY_STATUS secStatus =
    pSecurityFunctionTable->AcquireCredentialsHandle(
        nullptr,
        L"NTLM",
        SECPKG_CRED_OUTBOUND,
        nullptr,
        useAuth ? &auth : nullptr,
        nullptr,
        nullptr,
        &ctx->sspiWindowsHandles->credHandle,
        &expiry
    );
```

`useAuth` becomes true only for Negotiate when a username has been supplied explicitly. In the NTLM path, Qt passes `nullptr` rather than a `SEC_WINNT_AUTH_IDENTITY` structure to `AcquireCredentialsHandle()`. Under the SSPI model, this selects the default outbound credentials of the current Windows security context—typically the logged-on corporate account in a domain environment.

Qt does not obtain the password or NTLM hash in cleartext. SSPI returns an opaque credential handle from which it produces NTLM messages. This distinction matters: the vulnerability lets the attacker relay the victim's authentication exchange.

Stage 3 — The NTLM Context Is Not Bound to the Intended Proxy

The credential handle is passed to `qSspiContinue()`, where Qt continues the handshake through Windows SSPI:

qauthenticator.cpp qSspiContinue()

```
InitializeSecurityContext(
    &ctx->sspiWindowsHandles->credHandle,
    ...,
    ISC_REQ_ALLOCATE_MEMORY, // the only requested flag
    ...
);
```

Qt requests only automatic allocation for the output token. Three controls that could have hindered relay are absent:

- `ISC_REQ_INTEGRITY` is not set, so Qt does not require context integrity;
- no `SECBUFFER_CHANNEL_BINDINGS` buffer is supplied, so EPA/Channel Binding does not bind the NTLM exchange to a protected channel;
- for direct NTLM, `targetNameW` remains empty, so no SPN binds the context to the intended service.

Together, these omissions mean the SSPI context is not tied to the proxy to which the user believes they are authenticating. In the tested configuration, Qt generated NTLM Type 1 and Type 3 messages for the domain user, and the controlling proxy could relay them to another corporate service in real time.

Impact and Attack Surface

The scope was substantial: any corporate Windows system on which a user was logged in with a domain account could become a potential entry point. This was not an exotic laboratory configuration, but a widespread class of enterprise devices.

The practical result was an authenticated session on a relay-vulnerable target with the victim's permissions. Depending on the selected service and the victim's privileges, the consequences ranged from access to corporate data to remote code execution or compromise of domain infrastructure—all initiated by a single `407 Proxy Authentication Required` response.

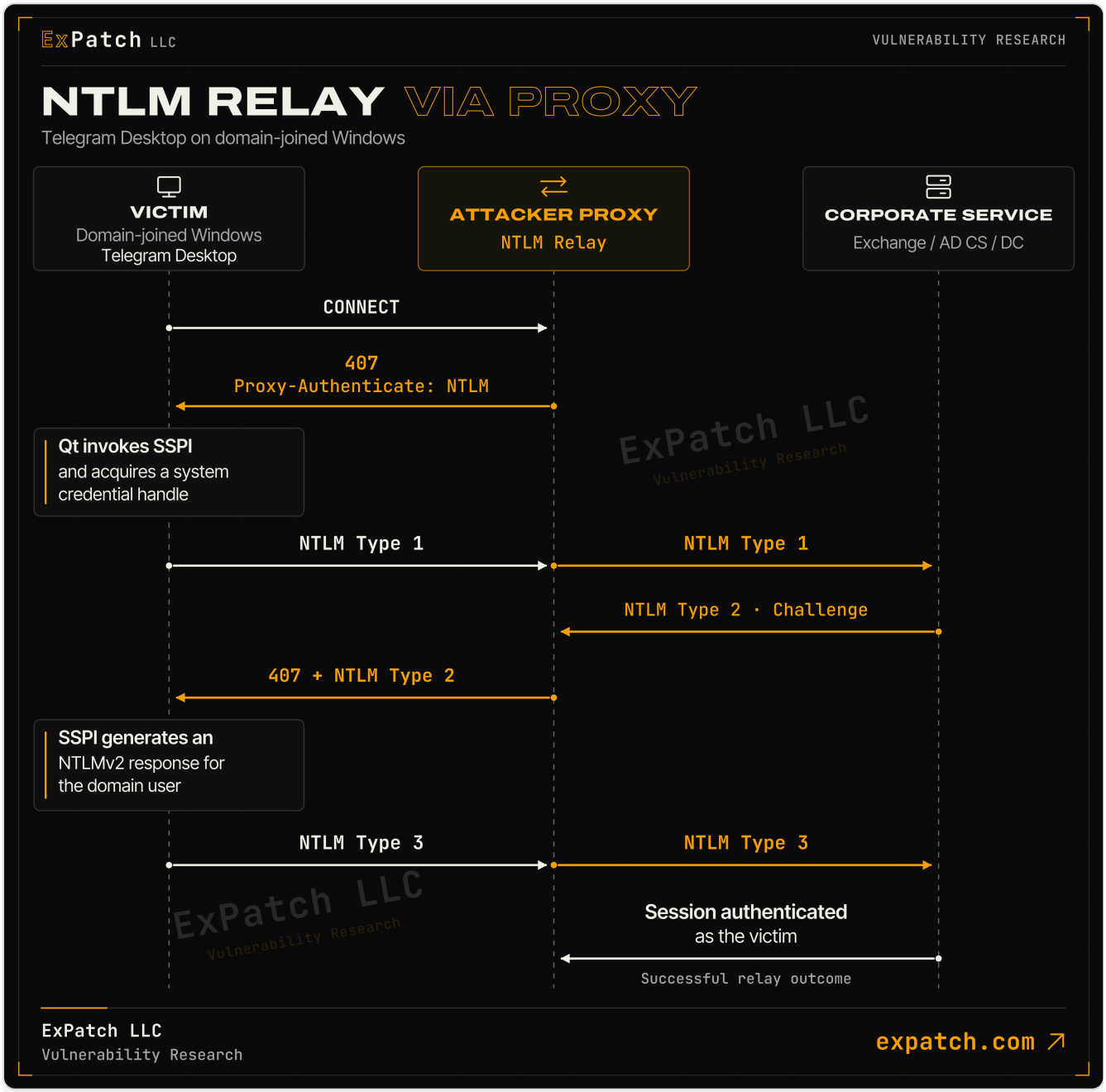


Fig. 2. *NTLM Relay: the proxy forwards NTLM messages between Telegram Desktop and a corporate service, obtaining an authenticated session with the victim's privileges.*

Two-Stage NTLM → Negotiate Escalation (P1): Two NTLMv2 Responses in One Session

Even supplying an explicit proxy username and password did not protect the user. A controlled server could force two consecutive authentications over a single connection:

Exchange sequence — NTLM → Negotiate

```
1. 407 + Proxy-Authenticate: NTLM
   |
   └─► Qt uses the explicitly configured proxy credentials
       and produces NTLMv2 response no. 1

2. The proxy rejects the first authentication
   |
   └─► qhttpsocketengine resets the authenticator;
       user becomes empty

3. 407 + Proxy-Authenticate: Negotiate
   |
   └─► qSspiStartup() requests the default SSPI credentials;
       Negotiate falls back to NTLM;
       Qt produces NTLMv2 response no. 2
       for the Windows domain user
```

The attacker therefore received two different NTLMv2 responses in one session: the first based on the explicit proxy credentials, and the second based on the logged-on domain user's account. Both were under attacker control, and the domain user's response could be relayed immediately to a selected corporate service.

This scenario significantly expanded the affected population. The attack was not limited to clients with an empty username field: after rejecting the first authentication and forcing a second exchange through Negotiate, the proxy could reach system credentials even when the user had supplied separate proxy credentials.

SOCKS5-in-Tunnel Vector (V3): Injecting a 407 Into an Active Tunnel

Vector V3 did not require the victim to follow a `tg://proxy` link. It affected domain-joined Windows systems where an enterprise proxy was deployed centrally through GPO, WPAD, or a PAC file.

The key difference from the base vector was the injection point. After a SOCKS5 tunnel had been established, the controlled proxy could see Telegram Desktop's unencrypted HTTP request—including its headers, URL, and MTProto POST body—and substitute the server response:

HTTP — response inside the SOCKS5 tunnel

```
HTTP/1.1 407 Proxy Authentication Required
Proxy-Authenticate: NTLM
```

Qt interpreted the response as a normal proxy-authentication challenge and entered the SSPI path automatically.

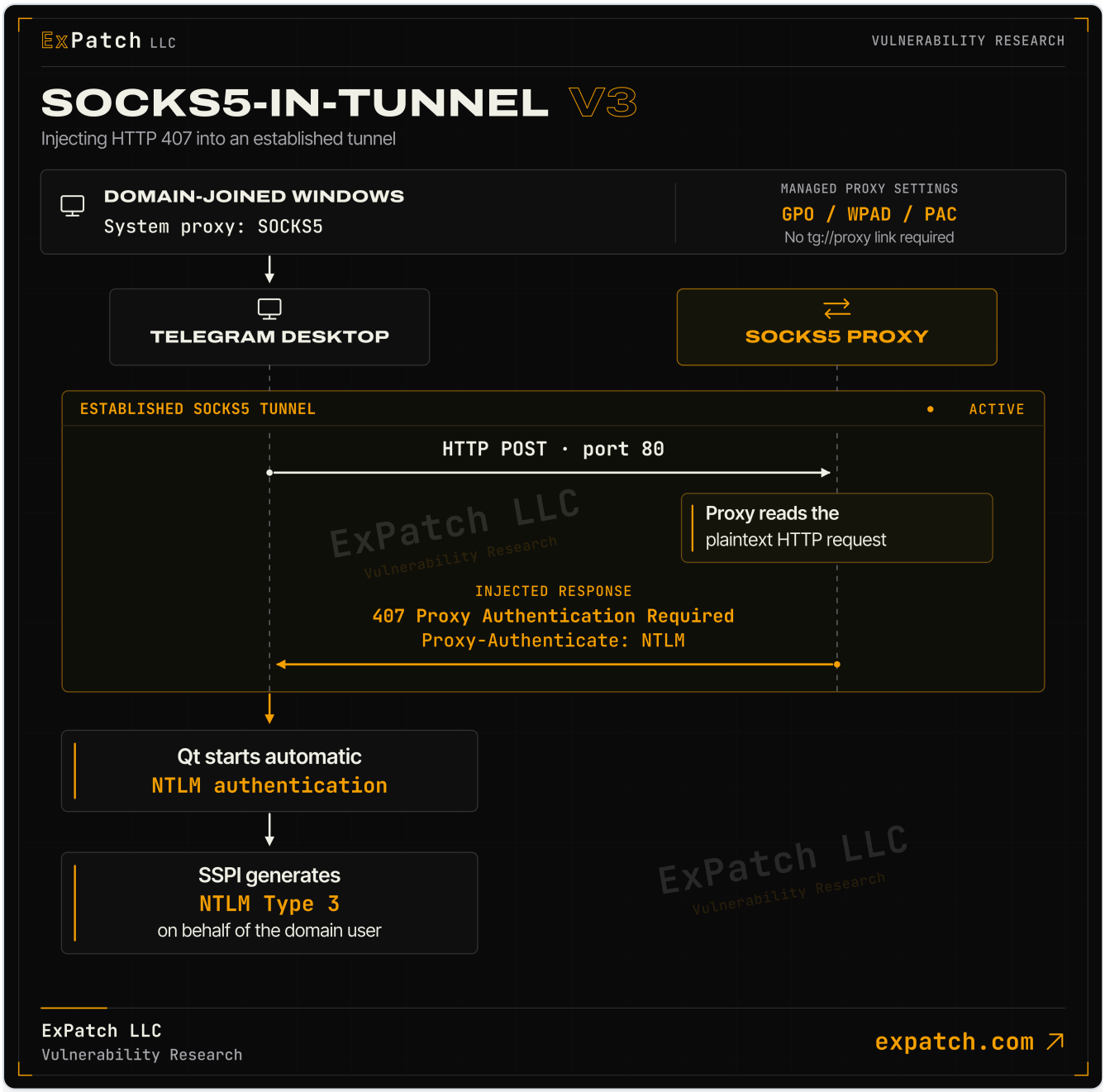


Fig. 3. *Vector V3: a controlled SOCKS5 proxy injects `407 Proxy Authentication Required` into an active tunnel, after which Qt automatically obtains an NTLMv2 response for the Windows domain user through SSPI.*

In a laboratory environment using a domain-joined Windows 10 machine, a single Telegram Desktop session produced more than 230 NTLM Type 3 messages. Qt's repeated attempts multiplied the amount of authentication material available to the attacker and the number of relay opportunities.

The Root Cause Is in QtNetwork

Further analysis showed that Telegram Desktop was only one route into the vulnerable code. The root cause was located in QtNetwork's shared HTTP authentication implementation, `qauthenticator.cpp`, and was not limited to `407 Proxy Authentication Required` responses.

The same path could be reached without any proxy. If an attacker-controlled HTTP server answered an ordinary unencrypted request with `401 Unauthorized` and the following header:

```http
WWW-Authenticate: NTLM
```

Qt treated it as a routine authentication challenge. On a domain-joined Windows system with no explicit credentials supplied, the library called SSPI without user interaction and began an NTLM exchange on behalf of the current account.

The proxy was therefore not a prerequisite for exploitation, but an especially dangerous entry point in Telegram Desktop. The vulnerable mechanism lived deeper in the common Qt HTTP client stack used by many applications.

Where Else the Vulnerable Code Was Reachable

We did not stop at source review of Qt and Telegram Desktop. This was not a theoretical extrapolation from a shared API: we reproduced the same vulnerable primitive in qBittorrent, Nextcloud Desktop, and Electrum.

Other Applications Tested

- **qBittorrent 4.6.3 with Qt 6.4.2.** A controlled proxy switched the application from Basic to NTLM with a `407 Proxy Authentication Required` response, without warning the user. The completed NTLM handshake produced a 65,076-byte `NtChallengeResponse`, approximately 64,876 bytes of which contained unrelated heap data.
- **Nextcloud Desktop 3.11.0 with Qt 5.15.13.** One test session performed four consecutive NTLM exchanges. Each response reached 65,076 bytes, disclosing roughly 260 KB of memory in total. The dumps contained complete HTTP requests, Nextcloud WebDAV and API routes, service headers, and NTLM tokens from previous exchanges.
- **Electrum 4.7.2 with Qt 6.11.0.** We found a separate direct path that required no proxy. A controlled Bitcoin or LNURL invoice description was rendered as rich text by a QML component; an injected `<img src="http://…">` caused `QNetworkAccessManager` to request an attacker-controlled server. A `401 WWW-Authenticate: NTLM` response made Qt send an unsigned NTLM Type 1 automatically, while Electrum had no `authenticationRequired` handler capable of stopping the exchange. We experimentally confirmed reachability and automatic NTLM initiation. Completing Type 3 through SSPI and relaying it requires a domain-joined Windows machine.

The same QtNetwork defect was therefore reproduced across independent products, Qt versions, and both 401 and 407 entry points. Telegram remained the principal object of the research, but the actual blast radius of the underlying root cause extended far beyond one application.

Second Primitive: Heap Memory Disclosure

Although the problem extended beyond Telegram, Telegram Desktop remained the primary target. We continued beyond forced NTLM authentication and relay. If an untrusted server could activate rarely used authentication code without user interaction and control the contents of NTLM Type 2, the remaining attacker-controlled fields also had to be audited.

That work exposed a second independent primitive: reading uninitialized heap memory in `qExtractServerTime()`:

qauthenticator.cpp qExtractServerTime()

```
1  static QByteArray qExtractServerTime(const QByteArray &targetInfoBuff)
2  {
3      QByteArray timeArray;
4      QDataStream ds(targetInfoBuff);
5      ds.setByteOrder(QDataStream::LittleEndian);
6      quint16 avId, avLen;
7
8      ds >> avId;
9      ds >> avLen;
10     while (avId != 0) {
11         if (avId == AVTIMESTAMP) {
12             timeArray.resize(avLen);                // server-controlled length
13             ds.readRawData(timeArray.data(), avLen); // short read is not checked
14         }
15         // ...
16     }
17 }
```

In NTLM Type 2, the server controls `targetInfo`, including the `MsvAvTimestamp` identifier and the 16-bit `avLen` field. We declared a length of 65,000 bytes but supplied only the eight-byte timestamp. Qt allocated the declared buffer, filled only the eight available bytes, and failed to check that `readRawData()` had performed a short read. The remaining 64,992 bytes retained prior heap contents. Qt then included the entire buffer—uninitialized area included—in the NTLMv2 response sent to the attacker's server.

An ordinary NTLM exchange thus became a remote, repeatable memory-read primitive. In captured dumps, we confirmed disclosure of code pointers and heap metadata sufficient to bypass ASLR, network addresses, VPN client and adapter names, HTTP headers, and previous NTLM tokens. One hundred exchanges over approximately five minutes yielded about 6.5 MB of Telegram Desktop process memory.

Fig. 4. *Leak mechanism: Qt allocates 65,000 bytes from attacker-controlled `avLen`, writes only the eight-byte timestamp, and includes the remaining 64,992 bytes of heap memory in NTLM Type 3.*

What the Leaked Memory Contained

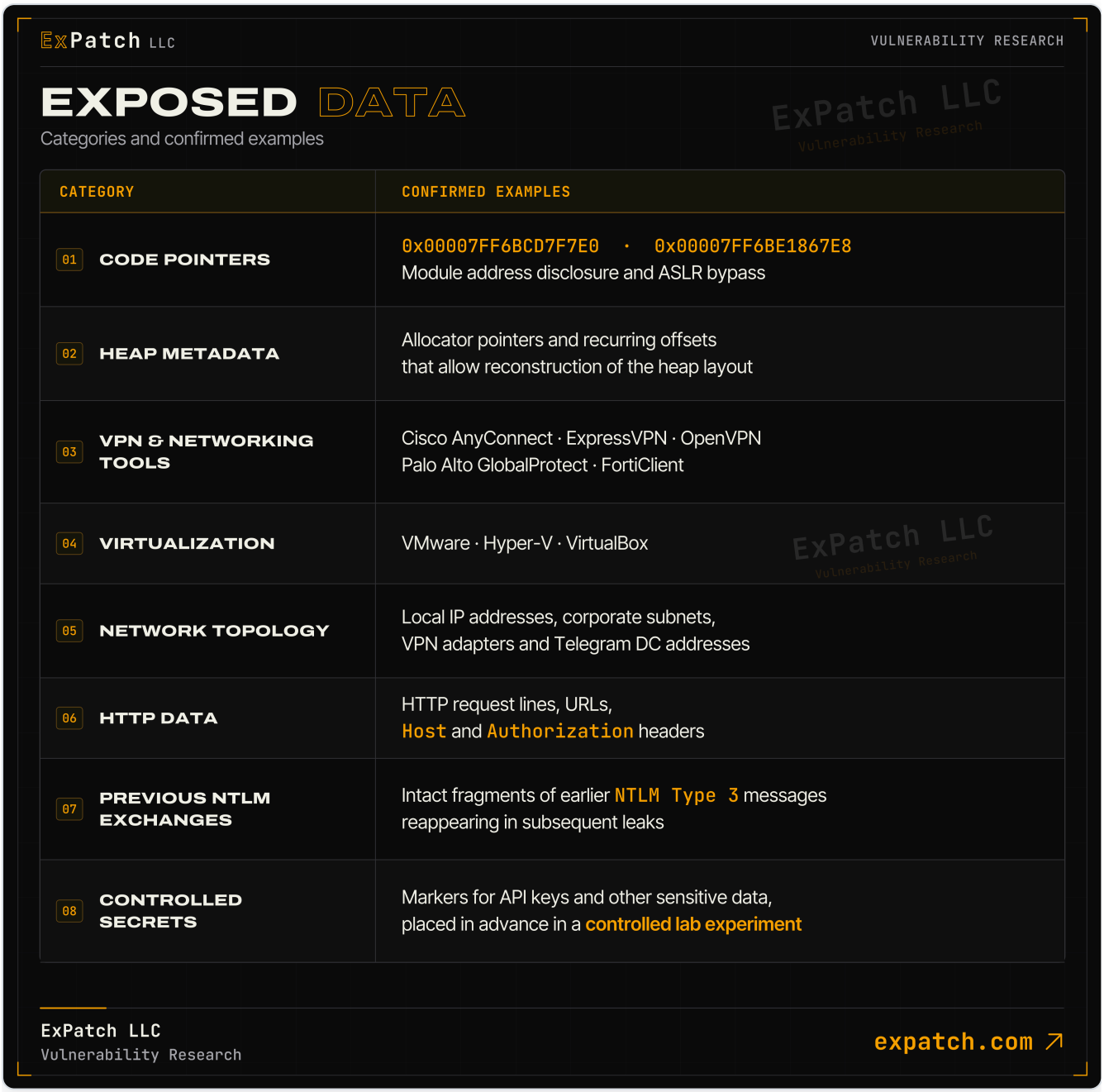


Fig. 5. *Categories of data found in leaked memory during laboratory testing.*

Individually, the fragments could look like random noise. Together, they formed a detailed profile of the device and its owner: username, work domain and hostname from NTLM, VPN software, virtualization, internal addressing, and enterprise network infrastructure.

The consequences were not limited to deanonymization. The vulnerability indiscriminately transmitted contents of reused heap buffers. Through controlled heap grooming, we placed test passwords and API-secret markers in those buffers and later recovered them from the NTLMv2 response sent to the proxy. The same mechanism could disclose API keys, session tokens, cookies, Authorization headers, and other secrets recently present in process memory. Later disclosures also contained previously generated NTLM Type 3 messages.

Importantly, the domain user's password was not recovered directly from NTLM. Cleartext passwords could leak as contents of a reused buffer, independently of the NTLM protocol itself.

The most troubling aspect was the channel: a feature intended to help users bypass restrictions and retain access to information. The proxy continued to function, Telegram displayed no warning, and the user could not see that a malicious proxy operator was receiving process-memory fragments—including passwords, API secrets, and data capable of identifying the user or granting access to connected systems.

Full RCE Chain: Laboratory Confirmation

To determine the maximum practical impact, we built an isolated Active Directory environment using Windows Server 2022 and an Enterprise CA with AD CS Web Enrollment enabled.

The victim ran the official Telegram Desktop 6.8.2 without any modification to its source, executable, Qt build, or authentication mechanisms. The client used its ordinary configuration. The user's only action was to add a proxy through Telegram's supported interface; every offensive component remained on the controlled proxy and laboratory infrastructure.

Once Telegram connected to the proxy, the chain executed automatically:

Fig. 6. *Complete laboratory chain: from adding a* `tg://proxy` *endpoint to obtaining SYSTEM on hosts where the compromised account held administrative privileges.*

All nine AD CS relay attempts authenticated successfully and issued a certificate. Because the test account had Domain Admin privileges, we obtained `NT AUTHORITY\SYSTEM` on both the workstation and the domain controller. The complete chain took approximately 30–35 seconds and required neither password guessing nor password recovery.

Post-compromise access was determined by the victim account's privileges: an ordinary user did not automatically become a domain administrator. Relaying an administrator, privileged employee, or service account, however, could lead directly to remote code execution and compromise of the corresponding systems. The experiment confirmed the worst-case impact demonstrated in our tests: complete domain takeover through an official, unmodified Telegram Desktop client after adding an attacker-controlled proxy.

The issued certificate also created persistence. A password change did not revoke it automatically; with a one- or two-year certificate lifetime, an attacker could continue authenticating until expiration, explicit revocation, or account disablement.

Why This Was a Telegram Desktop Vulnerability, Not Only a Qt Vulnerability

Reducing the issue to the source file containing the defect ignores the exploitation chain. The library root cause was indeed in QtNetwork. But the attack did not begin inside an abstract Qt application: it began in the official Telegram Desktop client after a proxy was added through the supported `tg://proxy` mechanism.

Telegram Desktop made the defect practically reachable by:

- accepting and persisting a user-supplied proxy;
- passing it to Qt without prohibiting NTLM and Negotiate;
- giving the user no opportunity to see or reject the proxy-selected scheme;
- allowing an untrusted proxy to activate SSPI and obtain an NTLMv2 response;
- transmitting Telegram Desktop process-memory fragments through the same channel.

This was more than the passive presence of a vulnerable dependency. Remove any key product element—`tg://proxy`, automatic proxy persistence, or the absence of an NTLM prohibition—and the Telegram path no longer works. Telegram's product decisions were therefore a necessary part of exploitation, not incidental surroundings for a library flaw.

Responsible Disclosure and Timeline

June 6: Telegram Receives the Complete Report

After a night of continuous validation and preparation, on June 6, 2026, we sent Telegram not a short description or preliminary hypothesis, but a finished 21-page technical report: EXPATCH-2026-TG-002. The package included source-code analysis, exploitation conditions, a PoC, a video demonstration, laboratory results, and the complete RCE chain.

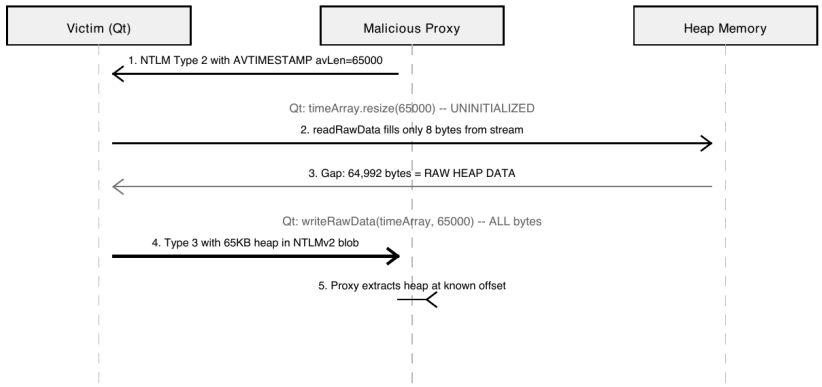
A dedicated section already documented the memory disclosure in `qExtractServerTime()`: the server-controlled `MsvAvTimestamp.avLen` field, expansion through `timeArray.resize(avLen)`, the unchecked short read in `readRawData()`, and transmission of the unread heap contents inside the NTLMv2 response. On page 20, we proposed a specific remediation: validate `avLen` against the remaining buffer and accept a timestamp only at its expected length of eight bytes.

65KB Heap Information Disclosure (F-QT-A14)

Each NTLM exchange leaks up to 65,535 bytes of uninitialized heap memory from the victim's Telegram Desktop process. The leaked data is sent DIRECTLY (not hashed) in the NTLMv2 response blob to the proxy. The proxy decodes base64 and reads raw heap bytes at a known offset.

How the Heap Leak Works

65KB Heap Leak Mechanism



Vulnerable Code Path (qauthenticator.cpp)

```
// Line 1330: Qt parses NTLM Type 2 targetInfo AV_PAIRs
if (avId == MsvAvTimestamp) {         // avId=7 from proxy

    // Line 1343: ALLOCATE -- avLen from proxy (uint16, up to 65535)
    QByteArray timeArray;
    timeArray.resize(avLen);           // <<< UNINITIALIZED HEAP!

    // Line 1345: READ -- fills only N bytes remaining in stream
    ds.readRawData(timeArray.data(), avLen);
    // Stream has ~8 bytes left. Gap = avLen - 8 = 64,992 bytes
    // Those 64,992 bytes contain WHATEVER WAS IN HEAP BEFORE

    // Line 1384: WRITE -- sends ALL bytes including uninitialized gap
    ds.writeRawData(timeArray.constData(), timeArray.size());
    // Entire timeArray goes into NTLMv2 response temp blob

    // Line 1417: SEND -- blob sent as base64 in Proxy-Authorization header
    ntChallengeResp.append(temp);
    // >>> 65KB of RAW HEAP sent to proxy in cleartext <<<
}
```

Proven Leaked Data (Live Telegram Desktop 6.8.2)

Data Category	Examples Found in Heap Dumps
Network Adapters	Intel(R) Wi-Fi 6E AX211 160MHz, VMware VMnet1/VMnet8
Virtualization	Hyper-V Virtual Ethernet Adapter, VirtualBox Host-Only
VPN Client Name	ExpressVPN Packet Filter Driver, OpenVPN Data Channel Offload
VPN Adapter Config	Cisco AnyConnect, Palo Alto GlobalProtect, FortiClient SSL VPN
Local IPs (private)	192.168.0.1, 192.168.0.6, 10.0.0.1, 172.19.0.1, 127.0.0.1
Public/External IPs	Telegram DC IPs: 149.154.167.41, 91.108.56.172

Fig. 7. *Page 5 of EXPATCH-2026-TG-002. The report delivered on June 6 already identified the vulnerable function, attacker-controlled `avLen`, disclosure of 64,992 heap bytes, and results reproduced against the standard Telegram Desktop 6.8.2 client.*

Expatch Security <security@expatch.llc> Jun 6, 2026, 6:11 AM

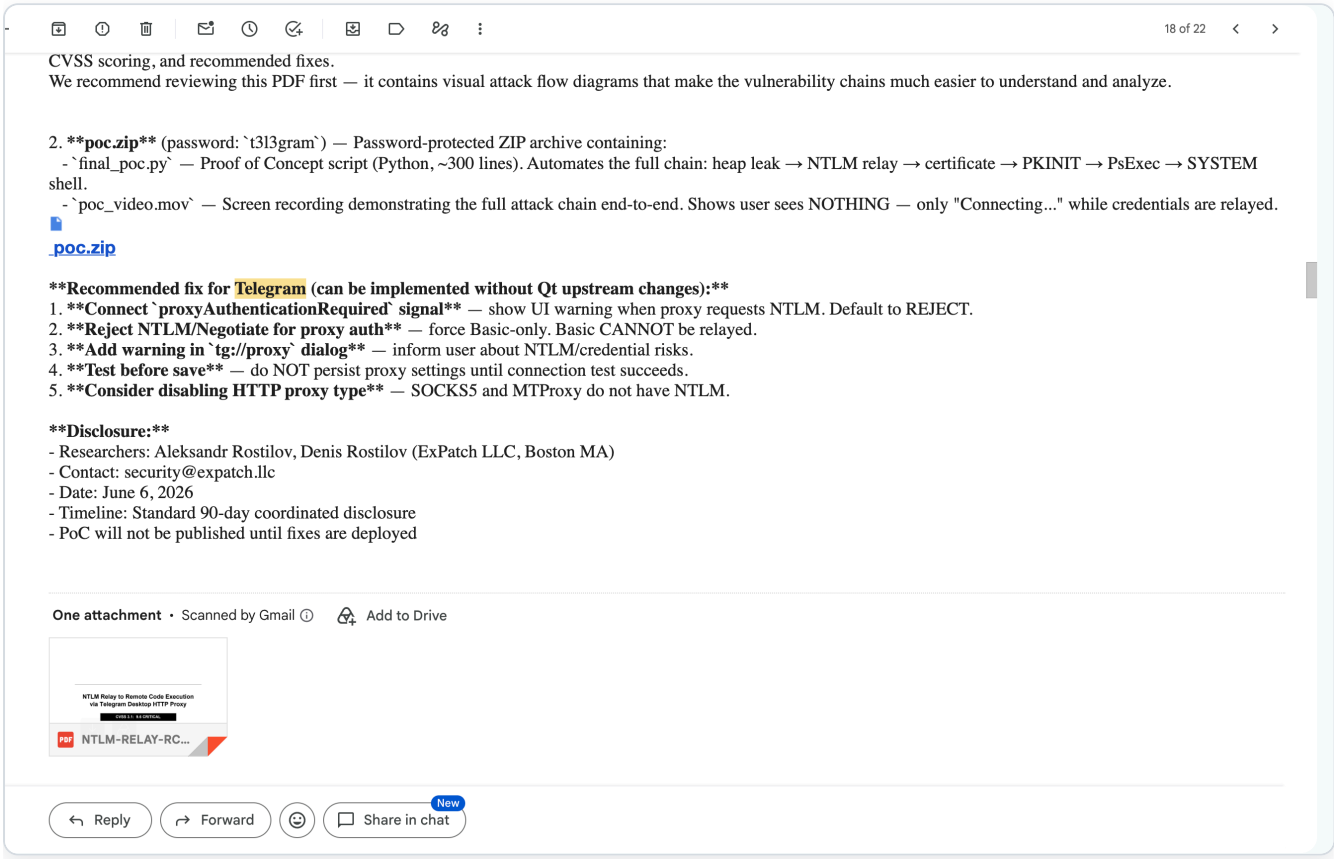
to Telegram

 # Attack surface: Telegram Desktop HTTP Proxy Authentication (Qt qauthenticator.cpp)
 # Severity: 9.6 (CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:H)

Description:

A chain of vulnerabilities in Qt's HTTP proxy authentication implementation allows a malicious proxy operator to silently relay a Telegram Desktop user's NTLM credentials to any NTLM-protected corporate service (Exchange, ADFS, VPN, ADCS, SMB, LDAP), achieving authenticated access as the victim without knowing their password.

Fig. 8. *The beginning of ExPatch's original June 6 disclosure to Telegram.*



18 of 22

CVSS scoring, and recommended fixes.
We recommend reviewing this PDF first — it contains visual attack flow diagrams that make the vulnerability chains much easier to understand and analyze.

2. **poc.zip** (password: `t3l3gram`) — Password-protected ZIP archive containing:
  - `final_poc.py` — Proof of Concept script (Python, ~300 lines). Automates the full chain: heap leak → NTLM relay → certificate → PKINIT → PsExec → SYSTEM shell.
  - `poc_video.mov` — Screen recording demonstrating the full attack chain end-to-end. Shows user sees NOTHING — only "Connecting..." while credentials are relayed.

poc.zip

Recommended fix for Telegram (can be implemented without Qt upstream changes):
1. **Connect `proxyAuthenticationRequired` signal** — show UI warning when proxy requests NTLM. Default to REJECT.
2. **Reject NTLM/Negotiate for proxy auth** — force Basic-only. Basic CANNOT be relayed.
3. **Add warning in `tg://proxy` dialog** — inform user about NTLM/credential risks.
4. **Test before save** — do NOT persist proxy settings until connection test succeeds.
5. **Consider disabling HTTP proxy type** — SOCKS5 and MTProxy do not have NTLM.

Disclosure:
- Researchers: Aleksandr Rostilov, Denis Rostilov (ExPatch LLC, Boston MA)
- Contact: security@expatch.llc
- Date: June 6, 2026
- Timeline: Standard 90-day coordinated disclosure
- PoC will not be published until fixes are deployed

One attachment • Scanned by Gmail Add to Drive

NTLM Relay to Remote Code Execution via Telegram Desktop HTTP Proxy

NTLM-RELAY-RC...

Reply Forward Share in chat New

Fig. 9. *Continuation of the June 6 email: delivered materials, mitigations Telegram could implement without waiting for upstream, researcher names, and the attached report.*

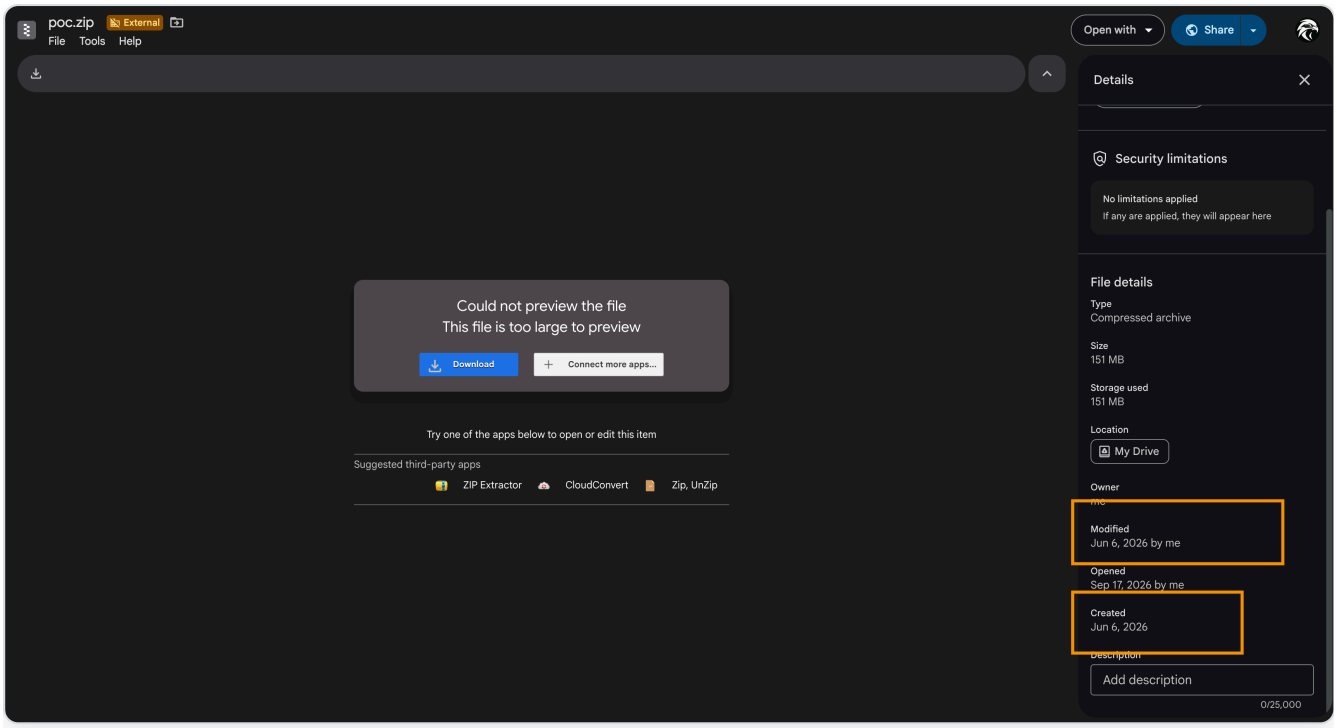


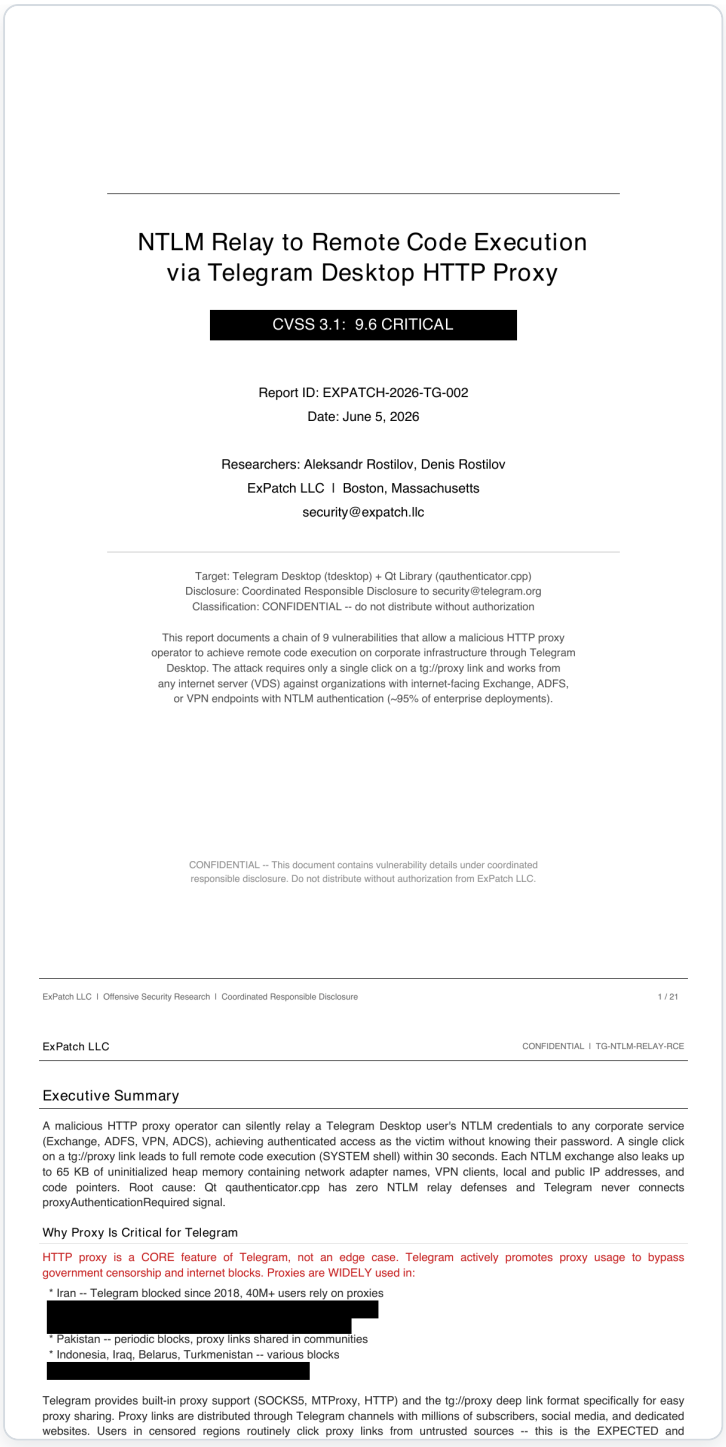
Fig. 10. *Google Drive metadata dates the creation and last modification of the delivered `poc.zip` archive to June 6, 2026.*

The image alone does not prove that the recipient downloaded the archive. Together with the email containing the `poc.zip` link, however, it establishes that the complete PoC existed and was available to Telegram on the day of disclosure.

The timeline is also supported by the original PDF metadata. Its cover is dated June 5, while `CreationDate` records June 6, 2026 at 04:04:01 UTC—approximately six hours before the email shown above. The file's SHA-256 is:

> **SHA-256**
>
> `c2c88a4db92a9fb06cd64a2efec3e337714a0ecdc32526c6a65b4dc477173f6c`

The cover prominently marked the document **CONFIDENTIAL** and stated that it was supplied under Coordinated Responsible Disclosure, with redistribution prohibited without permission from ExPatch LLC. We approached Telegram before completing a separate disclosure to Qt because Telegram could immediately block the most dangerous path inside its own application.



NTLM Relay to Remote Code Execution
via Telegram Desktop HTTP Proxy

CVSS 3.1: 9.6 CRITICAL

Report ID: EXPATCH-2026-TG-002
Date: June 5, 2026

Researchers: Aleksandr Rostilov, Denis Rostilov
ExPatch LLC | Boston, Massachusetts
security@expatch.llc

Target: Telegram Desktop (tdesktop) + Qt Library (qauthenticator.cpp)
Disclosure: Coordinated Responsible Disclosure to security@telegram.org
Classification: CONFIDENTIAL -- do not distribute without authorization

This report documents a chain of 9 vulnerabilities that allow a malicious HTTP proxy operator to achieve remote code execution on corporate infrastructure through Telegram Desktop. The attack requires only a single click on a tg://proxy link and works from any internet server (VDS) against organizations with internet-facing Exchange, ADFS, or VPN endpoints with NTLM authentication (~95% of enterprise deployments).

CONFIDENTIAL -- This document contains vulnerability details under coordinated responsible disclosure. Do not distribute without authorization from ExPatch LLC.

ExPatch LLC | Offensive Security Research | Coordinated Responsible Disclosure 1 / 21

ExPatch LLC CONFIDENTIAL | TG-NTLM-RELAY-RCE

Executive Summary

A malicious HTTP proxy operator can silently relay a Telegram Desktop user's NTLM credentials to any corporate service (Exchange, ADFS, VPN, ADCS), achieving authenticated access as the victim without knowing their password. A single click on a tg://proxy link leads to full remote code execution (SYSTEM shell) within 30 seconds. Each NTLM exchange also leaks up to 65 KB of uninitialized heap memory containing network adapter names, VPN clients, local and public IP addresses, and code pointers. Root cause: Qt qauthenticator.cpp has zero NTLM relay defenses and Telegram never connects proxyAuthenticationRequired signal.

Why Proxy Is Critical for Telegram

HTTP proxy is a CORE feature of Telegram, not an edge case. Telegram actively promotes proxy usage to bypass government censorship and internet blocks. Proxies are WIDELY used in:

* Iran -- Telegram blocked since 2018, 40M+ users rely on proxies
[redacted]
* Pakistan -- periodic blocks, proxy links shared in communities
* Indonesia, Iraq, Belarus, Turkmenistan -- various blocks
[redacted]

Telegram provides built-in proxy support (SOCKS5, MTProxy, HTTP) and the tg://proxy deep link format specifically for easy proxy sharing. Proxy links are distributed through Telegram channels with millions of subscribers, social media, and dedicated websites. Users in censored regions routinely click proxy links from untrusted sources -- this is the EXPECTED and

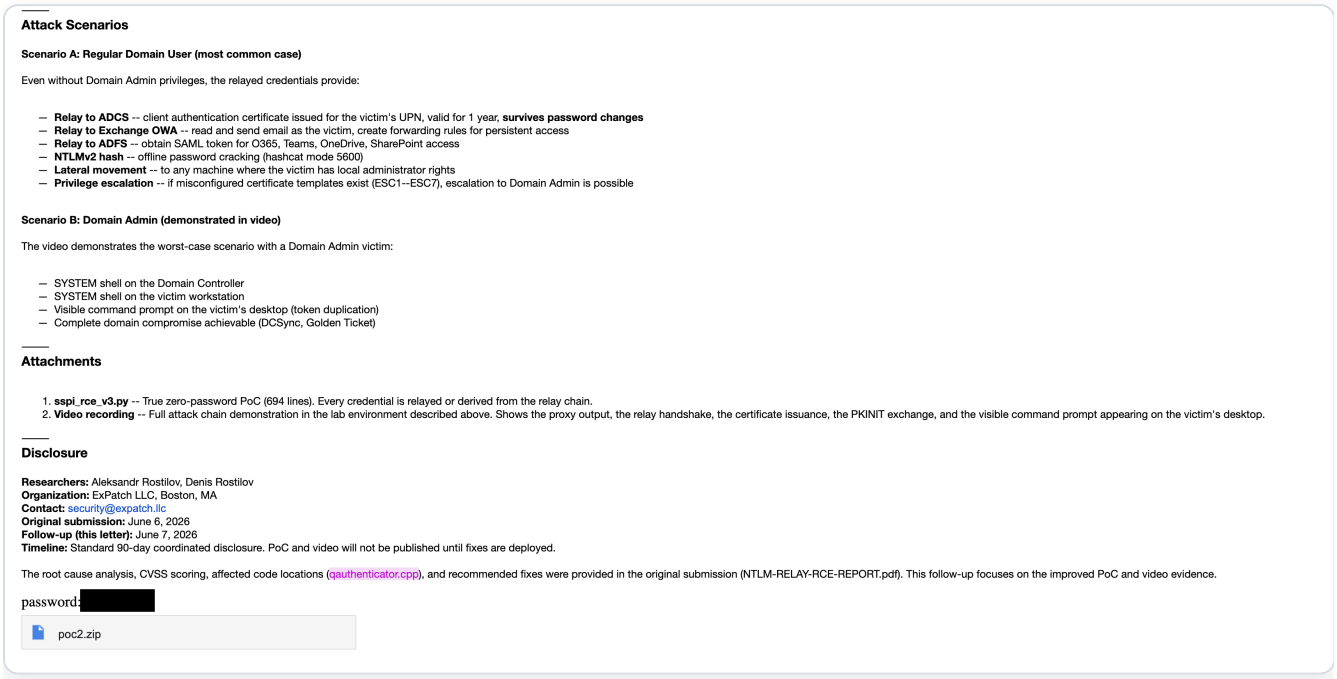
June 7: Telegram Receives a Fully Working Exploit

The original report was not the end of the disclosure. On June 7, we sent Telegram an additional email with an improved `sspi_rce_v3.py`: a 694-line end-to-end exploit requiring no password, accompanied by a video of the complete attack. This was neither a conceptual demonstration nor a loose collection of techniques; all necessary credentials were relayed automatically or extracted from the resulting relay chain.

The email described consequences for an ordinary domain user: client-certificate issuance through AD CS, access to Exchange OWA and ADFS, movement to machines where the victim had local administrative rights, and privilege escalation where certificate templates were insecure. In the Domain Admin scenario,

the demonstration ended with `NT AUTHORITY\SYSTEM` on both a workstation and the domain controller, from which complete domain compromise was possible.

The video showed the controlled proxy output, NTLM relay, certificate issuance, PKINIT authentication, and a command prompt appearing on the victim desktop. Telegram received not merely a description of potential harm, but a complete implementation reproducing the chain against a standard, unmodified Telegram Desktop client in an isolated environment.



Attack Scenarios

Scenario A: Regular Domain User (most common case)

Even without Domain Admin privileges, the relayed credentials provide:

- **Relay to ADCS** -- client authentication certificate issued for the victim's UPN, valid for 1 year, **survives password changes**
- **Relay to Exchange OWA** -- read and send email as the victim, create forwarding rules for persistent access
- **Relay to ADFS** -- obtain SAML token for O365, Teams, OneDrive, SharePoint access
- **NTLMv2 hash** -- offline password cracking (hashcat mode 5600)
- **Lateral movement** -- to any machine where the victim has local administrator rights
- **Privilege escalation** -- if misconfigured certificate templates exist (ESC1--ESC7), escalation to Domain Admin is possible

Scenario B: Domain Admin (demonstrated in video)

The video demonstrates the worst-case scenario with a Domain Admin victim:

- SYSTEM shell on the Domain Controller
- SYSTEM shell on the victim workstation
- Visible command prompt on the victim's desktop (token duplication)
- Complete domain compromise achievable (DCSync, Golden Ticket)

Attachments

1. **sspi_rce_v3.py** -- True zero-password PoC (694 lines). Every credential is relayed or derived from the relay chain.
2. **Video recording** -- Full attack chain demonstration in the lab environment described above. Shows the proxy output, the relay handshake, the certificate issuance, the PKINIT exchange, and the visible command prompt appearing on the victim's desktop.

Disclosure

Researchers: Aleksandr Rostilov, Denis Rostilov
Organization: ExPatch LLC, Boston, MA
Contact: security@expatch.llc
Original submission: June 6, 2026
Follow-up (this letter): June 7, 2026
Timeline: Standard 90-day coordinated disclosure. PoC and video will not be published until fixes are deployed.

The root cause analysis, CVSS scoring, affected code locations (qauthenticator.cpp), and recommended fixes were provided in the original submission (NTLM-RELAY-RCE-REPORT.pdf). This follow-up focuses on the improved PoC and video evidence.

password: [redacted]

poc2.zip

Fig. 12. *Additional June 7 disclosure: exploitation scenarios, the 694-line `sspi_rce_v3.py`, a video of the complete chain, and the attached `poc2.zip`. The email also records ExPatch's commitment not to release the PoC or video before fixes were available.*

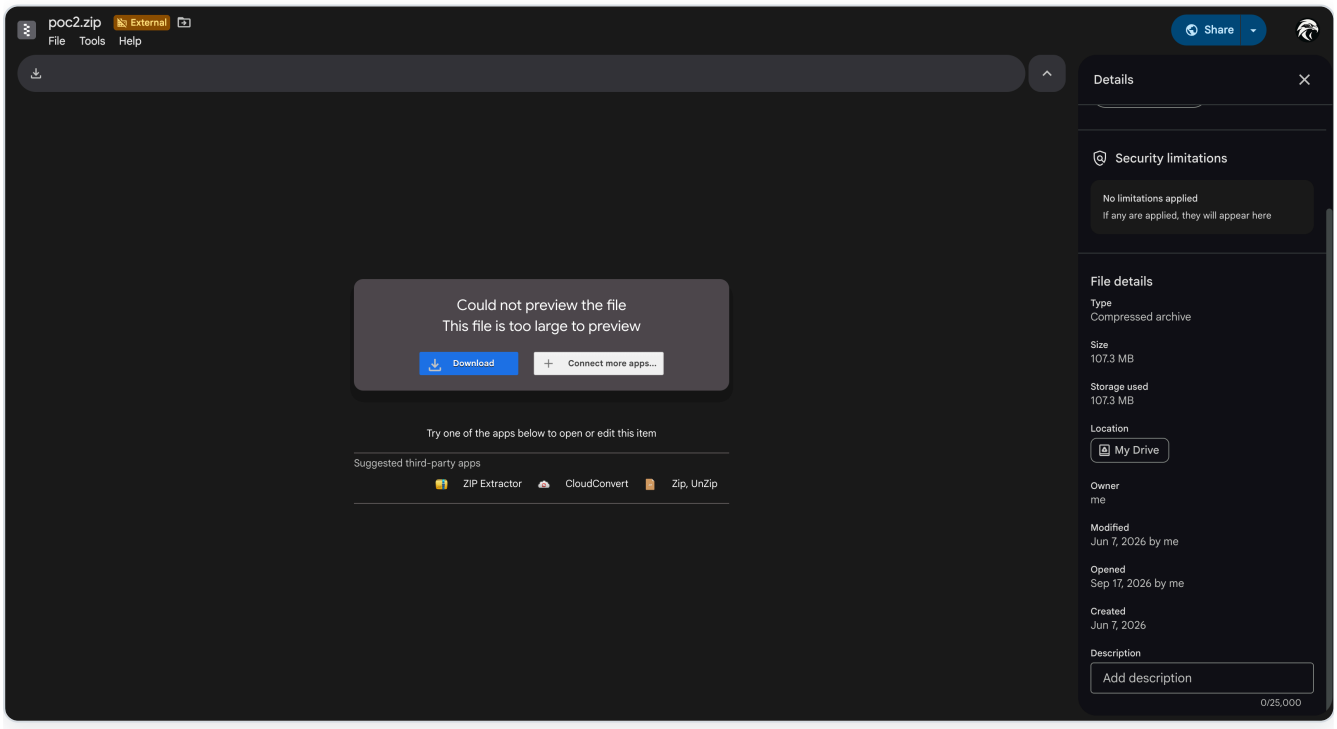


Fig. 13. *Google Drive records the creation and last modification of `poc2.zip` on June 7, 2026.*

June 8: Investigation Acknowledged and Two Precise Patches Appear

At 05:08 on June 8, Telegram Support acknowledged receipt and wrote:

> “The team is investigating your research, and we will contact you soon.”

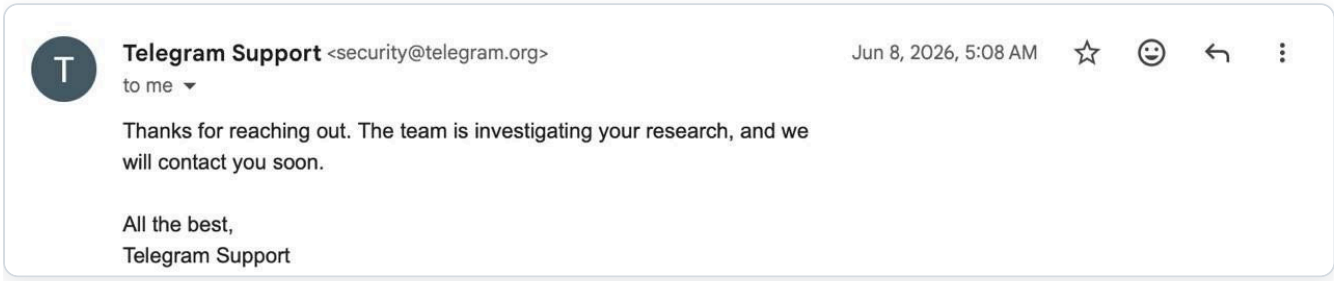


Fig. 14. *Telegram’s June 8 response confirming that the team was investigating the submitted research.*

The public history of the dependency repository used by Telegram Desktop dates two new commits to the same day:

- at 13:23:04 UTC, a Qt 5 patch;
- at 13:27:01 UTC, a Qt 6 patch.

Both commits identify John Preston as the author. In New York time, they were recorded at 09:23 and 09:27—approximately four hours after Telegram’s reply.

Their content matters as much as their timing. The Qt 6 patch introduced a new file, `qtbase_6.11.1/0038-disable-proxy-ntlm.patch`, and addressed both independent issues described in our report: forced NTLM/Negotiate authentication through a proxy and the memory disclosure in `qExtractServerTime()`.

Fig. 15. *Correspondence between the June 6 ExPatch report and the June 8 Telegram patch. Two independent findings match: disabling NTLM/Negotiate for proxy authentication and correcting* `MsvAvTimestamp.avLen` *handling.*

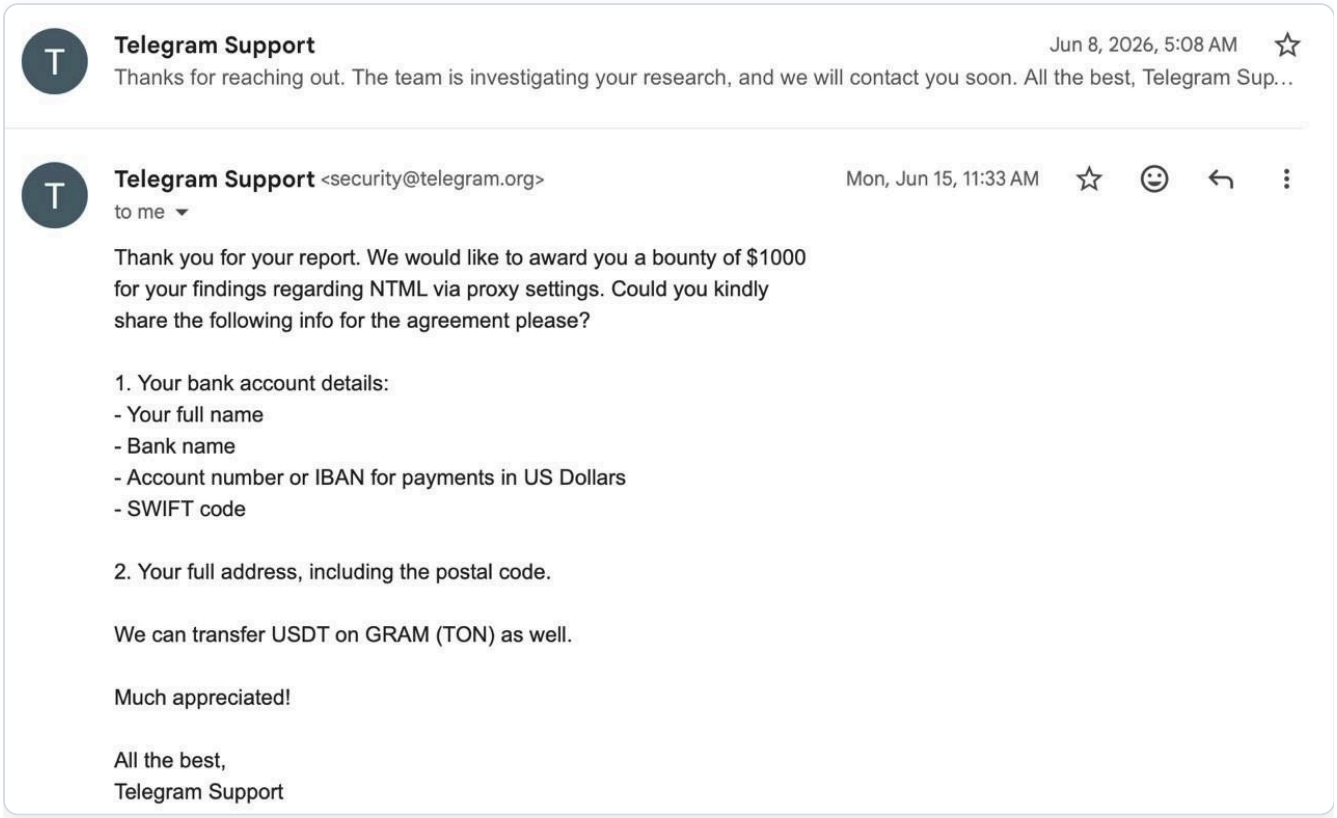
The change was not generic "NTLM hardening." It added an `avLen > end - p` remaining-buffer check, required `avLen == 8`, returned and wrote exactly eight timestamp bytes, and introduced `qNtlmBufferFits()` for `targetName` and `targetInfo` bounds checks. These are the same fields, conditions, and safeguards identified in the report.

The patch also covered both Qt network paths used by Telegram: `QHttpNetworkConnection` and `QHttpSocketEngine`. When NTLM or Negotiate was detected, the authenticator was reset, the connection ended with `ProxyAuthenticationRequiredError`, and no NTLMv2 response was sent to the proxy.

June 15–16: Bounty Offer, NDA, and Confirmation of the Fixes

On June 15, Telegram thanked us for the report and offered a $1,000 bounty:

> “We would like to award you a bounty of $1000 for your findings regarding NTLM via proxy settings.”



Telegram Support Jun 8, 2026, 5:08 AM
Thanks for reaching out. The team is investigating your research, and we will contact you soon. All the best, Telegram Sup…

Telegram Support <security@telegram.org> Mon, Jun 15, 11:33 AM
to me

Thank you for your report. We would like to award you a bounty of $1000 for your findings regarding NTML via proxy settings. Could you kindly share the following info for the agreement please?

1. Your bank account details:
- Your full name
- Bank name
- Account number or IBAN for payments in US Dollars
- SWIFT code

2. Your full address, including the postal code.

We can transfer USDT on GRAM (TON) as well.

Much appreciated!

All the best,
Telegram Support

Fig. 16. *Telegram’s June 15 offer. The bounty is described as covering findings related to NTLM through proxy settings.*

By then, Telegram’s patch contained two independent measures: prohibiting NTLM/Negotiate for proxies and fixing the `qExtractServerTime()` memory disclosure. The wording of the message did not establish whether the decision covered the complete set of scenarios or only the proxy-related portion of the research.

Money was not our primary concern. We asked Telegram to donate the offered amount to a charity of its choice and requested clarification on whether the decision was final for every reported vulnerability or only for NTLM disclosure through proxy settings.

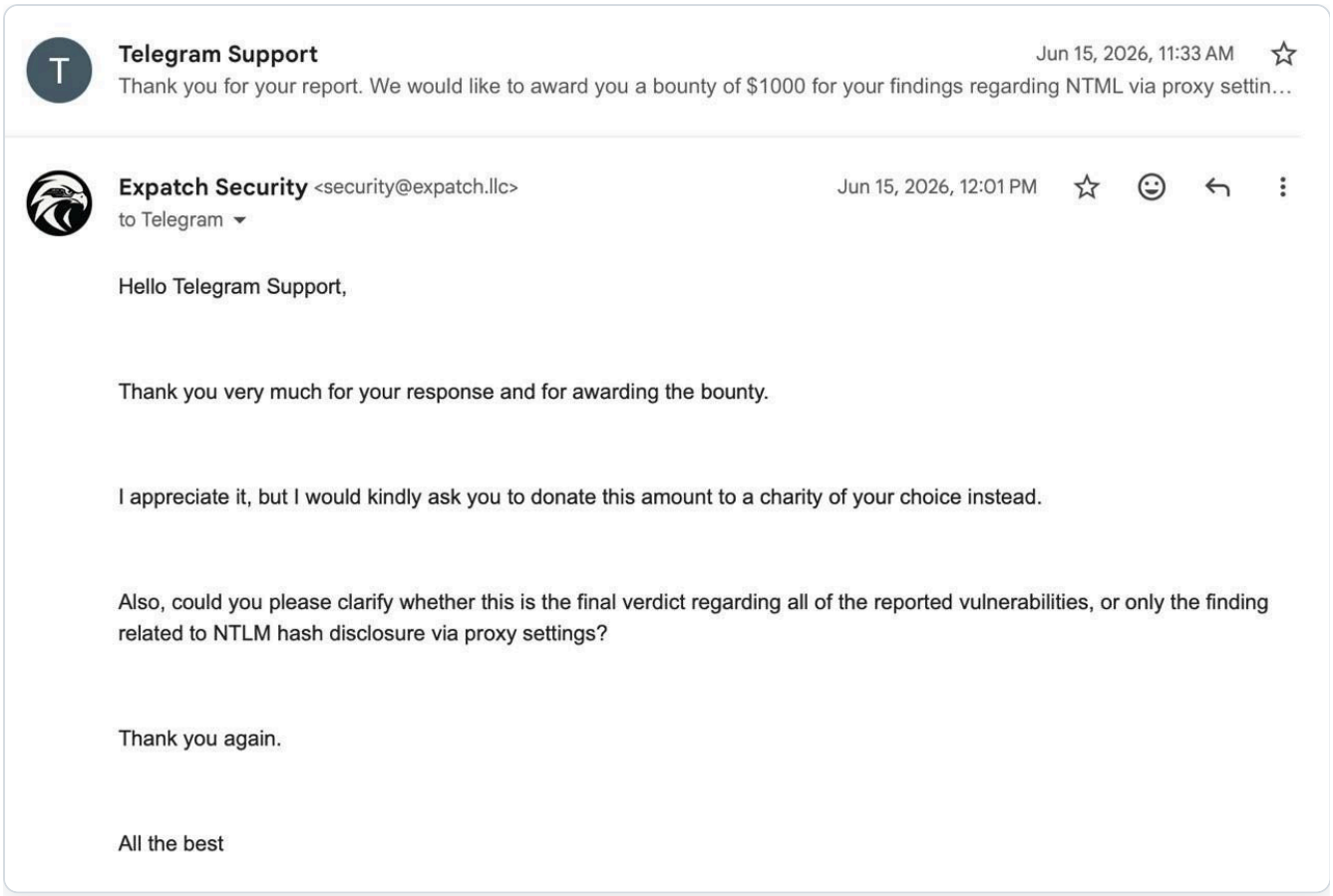


Fig. 17. *ExPatch's June 15 reply: a request to donate the bounty and clarify which findings the decision covered.*

At 05:07 on June 16, Telegram said the bounty decision was final and directly proposed an agreement that effectively functioned as an NDA. According to the message, its purpose was not only to process the payment or donation, but also to ensure that details of discovered issues were shared "responsibly and only with authorized parties."

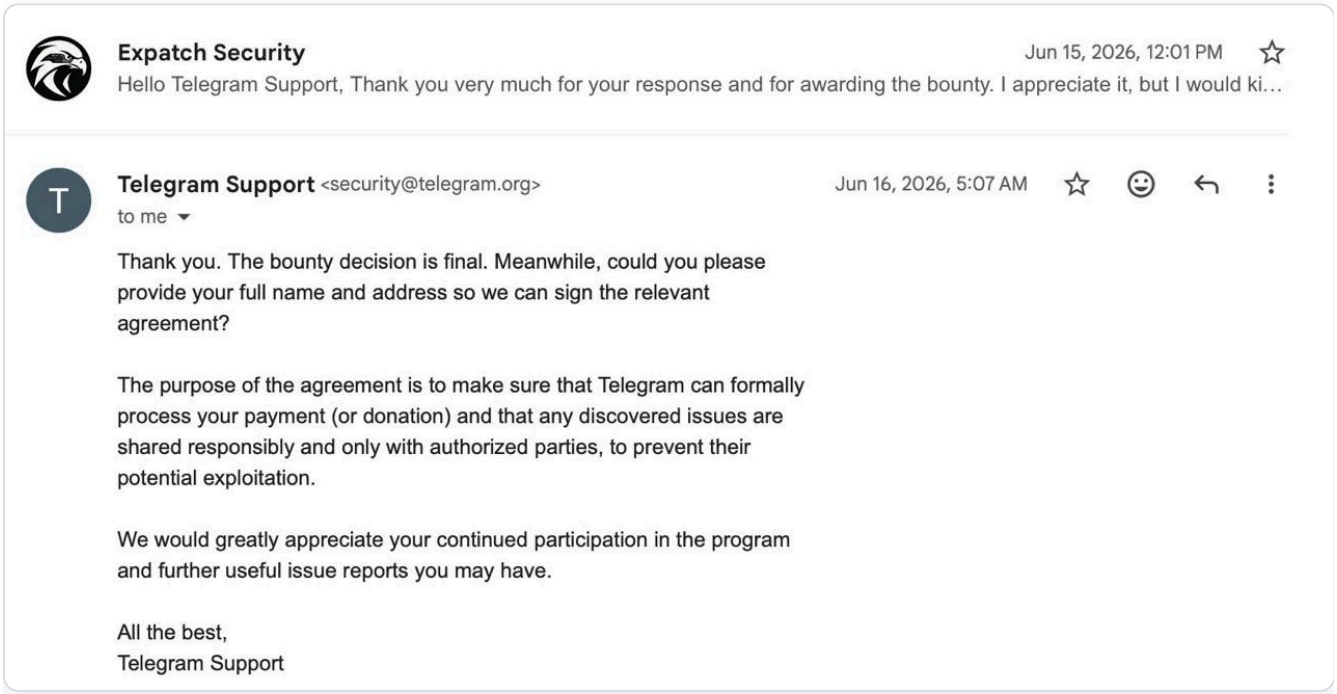


Fig. 18. *Telegram's June 16 response: an agreement linking payment or donation processing to restricting vulnerability information to authorized parties.*

At 05:56, we declined to sign the agreement and recorded our responsible-disclosure position: the technical report would be published after remediation or upon expiration of 90 days from the initial disclosure. We also asked Telegram to notify us once the issues had been fixed or otherwise mitigated.

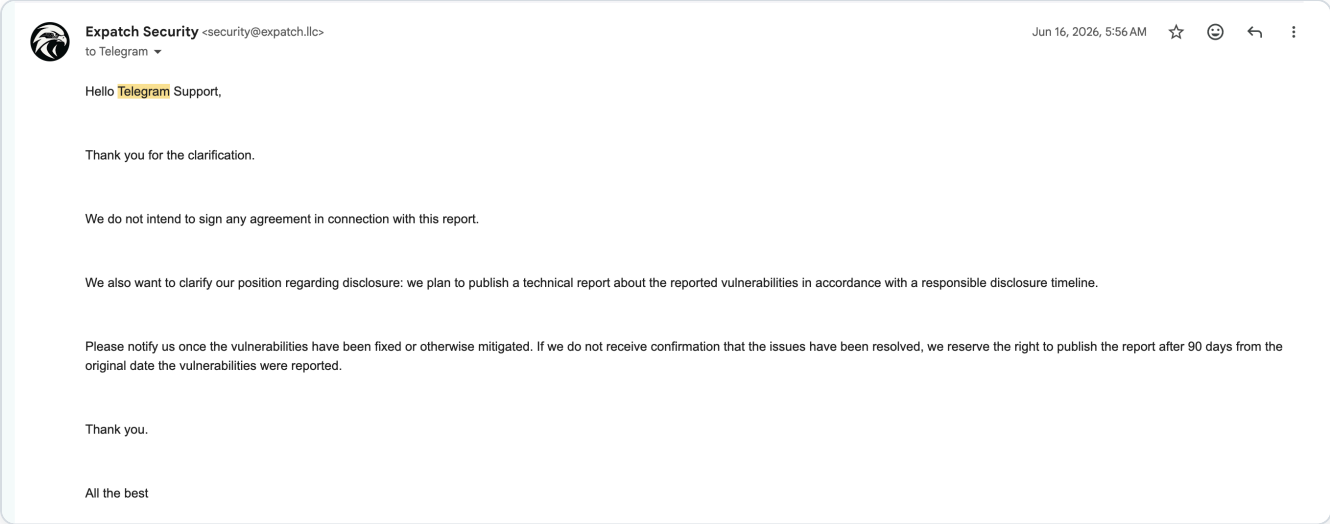


Fig. 19. *ExPatch's June 16 email: refusal of the agreement, confirmation of the 90-day disclosure window, and a request for remediation status.*

At 08:53, Telegram sent a separate technical clarification:

> "Telegram Desktop v6.9.2 contains fixes for all reported scenarios, with one exception."

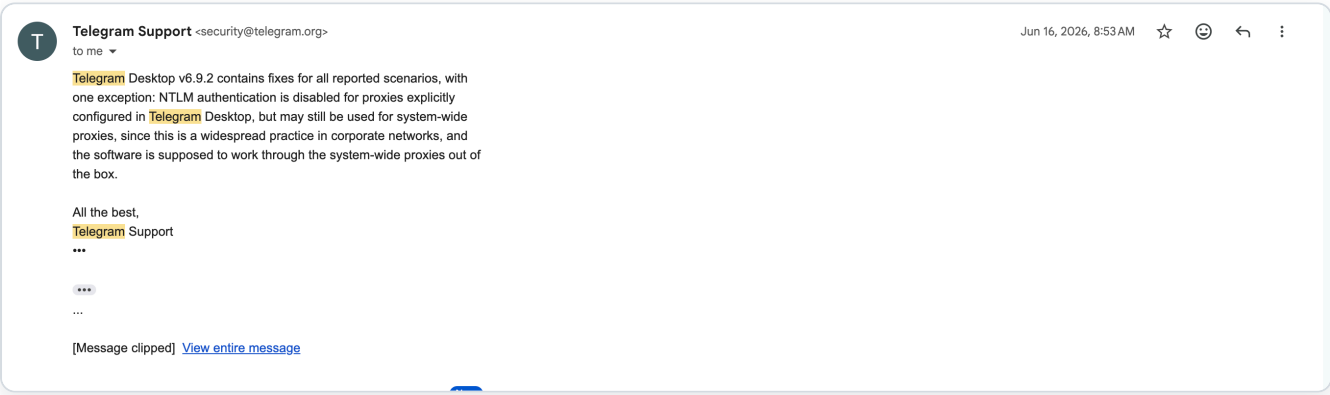


Fig. 20. *Telegram links version 6.9.2 to remediation of all submitted scenarios, with a specific exception for system-wide proxies.*

According to Telegram, NTLM was disabled for proxies explicitly configured inside Telegram Desktop, but remained available to system-wide proxies because those configurations are widely used in corporate networks. This exception exactly matched the distinction introduced while the patch was revised on June 10–11: proxies from `tg://proxy` and application settings were blocked, while system proxies were allowed to use NTLM.

After receiving this response, we decided to complete our separate disclosure to Qt and downloaded the current Qt 6.11.2 branch to check whether the issues were still reproducible. In that branch, however, the vulnerable paths had already been closed.

Reviewing Qt's history after Telegram's June 16 response, we discovered matching upstream fixes. Telegram had told us neither that the materials were being transferred nor that coordination with Qt was taking place. ExPatch and the researchers were excluded from the public record of the fixes.

What Matters Is Not the Number of Matches, but Their Precision

We do not claim every subsequent NTLM change in Qt. Our case does not require a long list of tasks and does not rest on broad thematic similarity. It is based on a documented sequence and correspondence traceable to specific lines of code.

On June 6, Telegram received our confidential report describing the `qExtractServerTime()` defect: the server-controlled `MsvAvTimestamp.avLen` field, expansion through `timeArray.resize(avLen)`, the incomplete input read, and the subsequent inclusion of uninitialized heap memory in the NTLMv2 response. The report also proposed validating `avLen` against the remaining buffer and accepting only the expected eight-byte timestamp.

On June 8, Telegram's repository history records a patch implementing those checks. It added `avLen > end - p` and `avLen != 8`, returned only the eight timestamp bytes, and limited the subsequent write to those same eight bytes. The patch simultaneously closed the second reported path by prohibiting NTLM/Negotiate for proxies configured explicitly inside Telegram Desktop.

On June 10, Qt opened QTBUG-147373, recording the start of upstream NTLM work two days after Telegram's acknowledgement. The associated commit was 6e88e0f708. Two days later, on June 12, Gerrit 744448 appeared: the same `qExtractServerTime()` function, the same `MsvAvTimestamp.avLen` field, an eight-byte timestamp requirement, and boundary-condition tests.

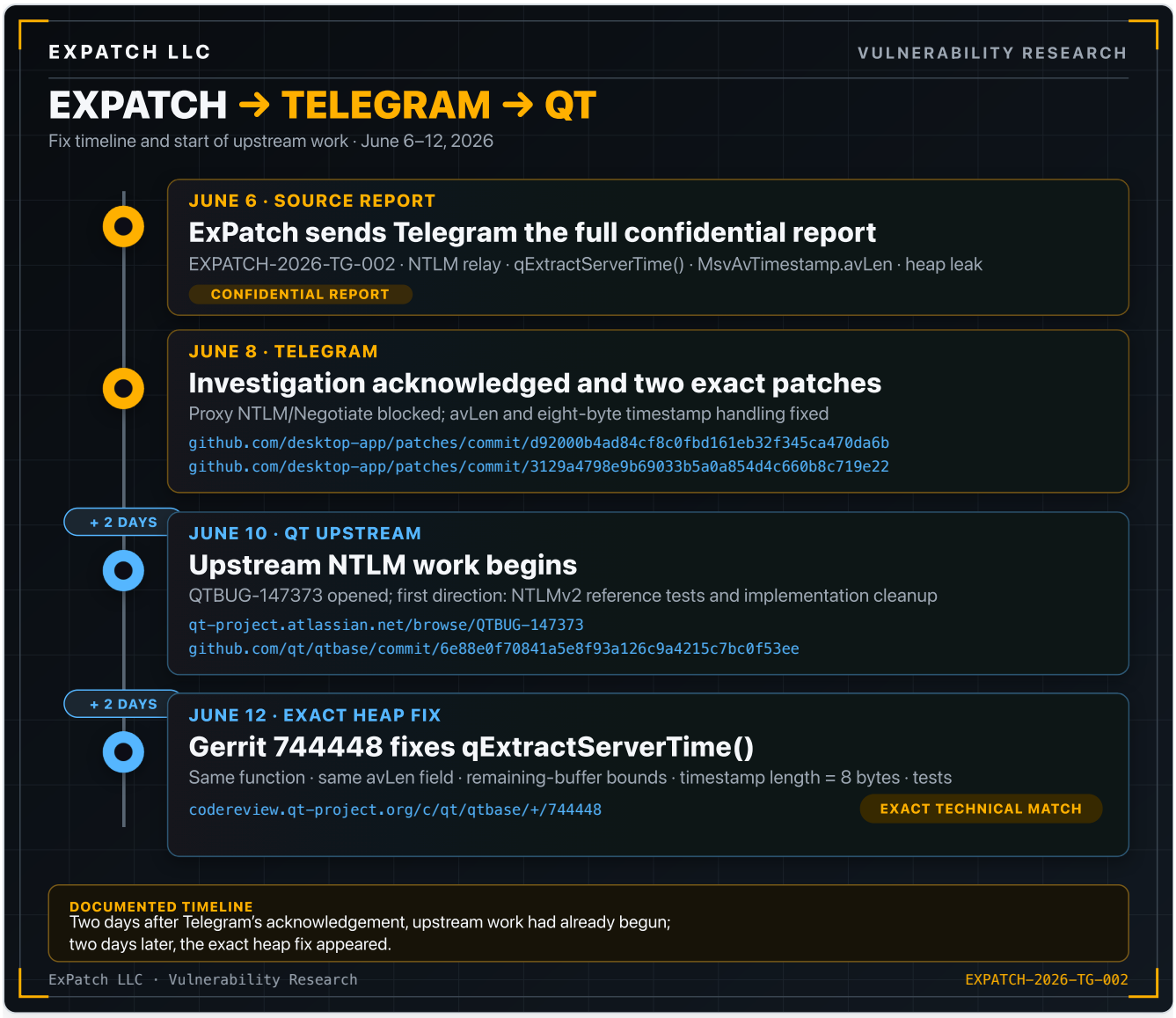


Fig. 21. *ExPatch's report and the corresponding Telegram and Qt changes: June 6–12, 2026.*

Telegram's patches addressed both problems described in our report: they prohibited NTLM/Negotiate for proxies added within the application and fixed the heap-memory disclosure. Qt subsequently fixed the same timestamp-handling flaw, checking the bounds of the input buffer and requiring an eight-byte timestamp. The match is traceable to the specific defect, the safeguards we proposed, and the behavior of the corrected code.

These changes addressed the problems at two levels. Telegram restricted the dangerous behavior within its own product; the Qt fix addressed the data-handling flaw in the shared library. That is why information about the upstream work mattered to us: we were preparing a separate disclosure to the Qt developers, only to discover that the corresponding changes had already been made. Users received protection, but the researchers were not included in the coordination behind it.

Neither Telegram nor Qt publicly explained the origin of the technical match between the report and the patches or credited us in the records we reviewed. The question is no longer whether a match exists, but how the results of our confidential research reached the upstream fix without notification or attribution.

This Is Not a Dispute Over One Thousand Dollars

For a small research team, public attribution is not a formality or a matter of ego. It is the professional result of forty days of work, evidence of competence, and the foundation of trust on which coordinated disclosure depends. A researcher gives a vendor unpublished material first, granting time and a technical advantage to protect users. In return, the researcher is entitled to expect transparent coordination and credit for their work.

Telegram used that head start to prepare fixes, and users benefited. But when the corresponding Qt changes appeared, we were not informed of the coordination, and neither ExPatch nor the researchers appeared in the public records we reviewed. The company received every benefit of responsible disclosure; the researchers did not receive even the basic transparency on which the model relies.

We are not asking readers to take our word for it. We are publishing the original report, its SHA-256 and metadata, correspondence, laboratory results, vulnerable code, patches, and an exact timeline. We expect Telegram and Qt to answer in the same language—the language of facts: how the results of our confidential research came to be reflected in the upstream fix, and why their source remained unattributed.

> Thank you for your time and attention. Research does not live where it is published, but where it is read, verified, and discussed. That is how an individual finding becomes a contribution to the security of every user.

> Sincerely,
> Alexander Rostilov and Denis Rostilov
> ExPatch Vulnerability Research Team