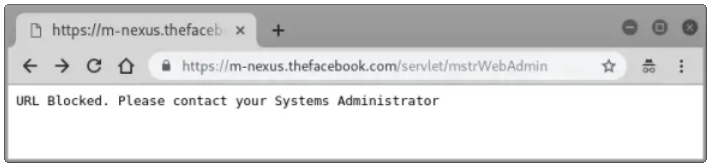
Bipin Jitiya Follow
May 31, 2020 · 11 min read · Listen

How I made $31500 by submitting a bug to Facebook

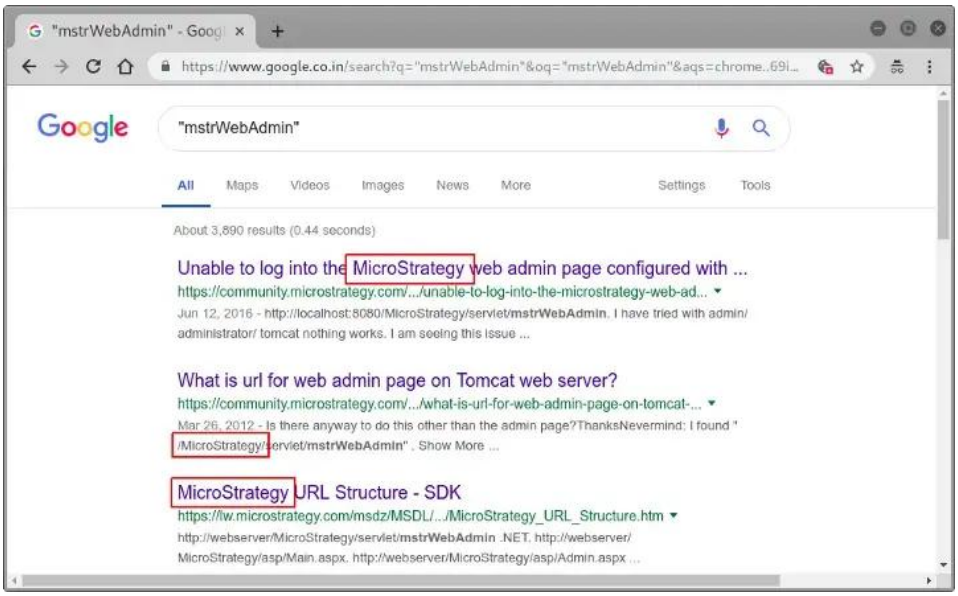
How did I found SSRF in Facebook — the story of my first bug bounty

Hello World ❤️,

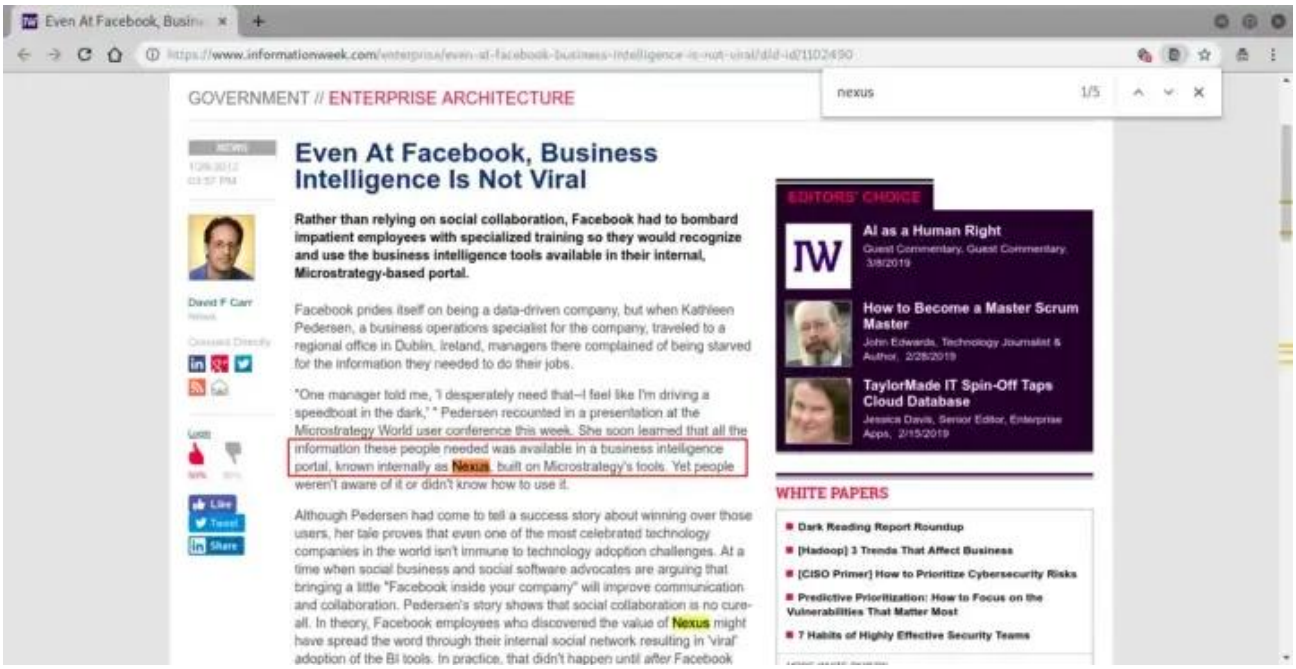
Facebook is the largest social networking site in the world and one of the most widely used. I have always been interested in testing the security of Facebook. During the sub domain enumeration, I've got a sub domain which is "**https://m-nexus.thefacebook.com/**". It redirects me to "https://m-nexus.thefacebook.com/servlet/mstrWebAdmin" observe below screenshot:



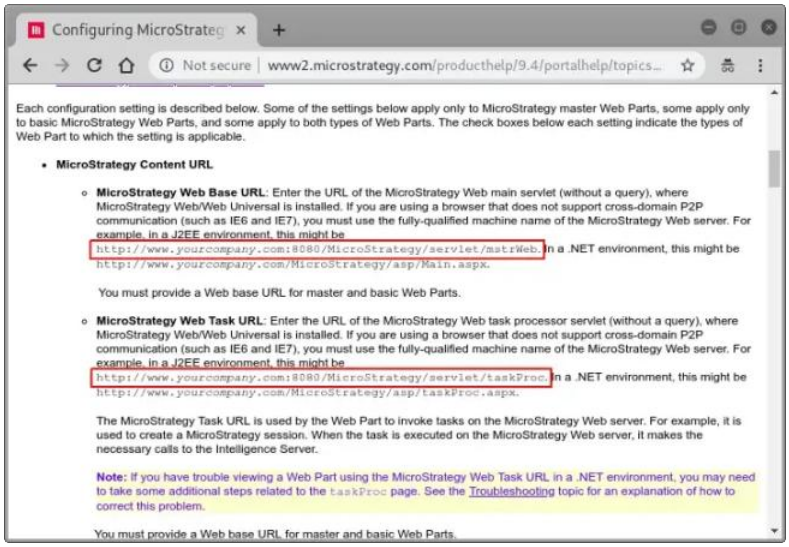
I quickly Google keyword **mstrWebAdmin** and I observed that this is the **Business Intelligence Portal** that is built on **MicroStrategy's** tools:



I confirmed it with a blog:

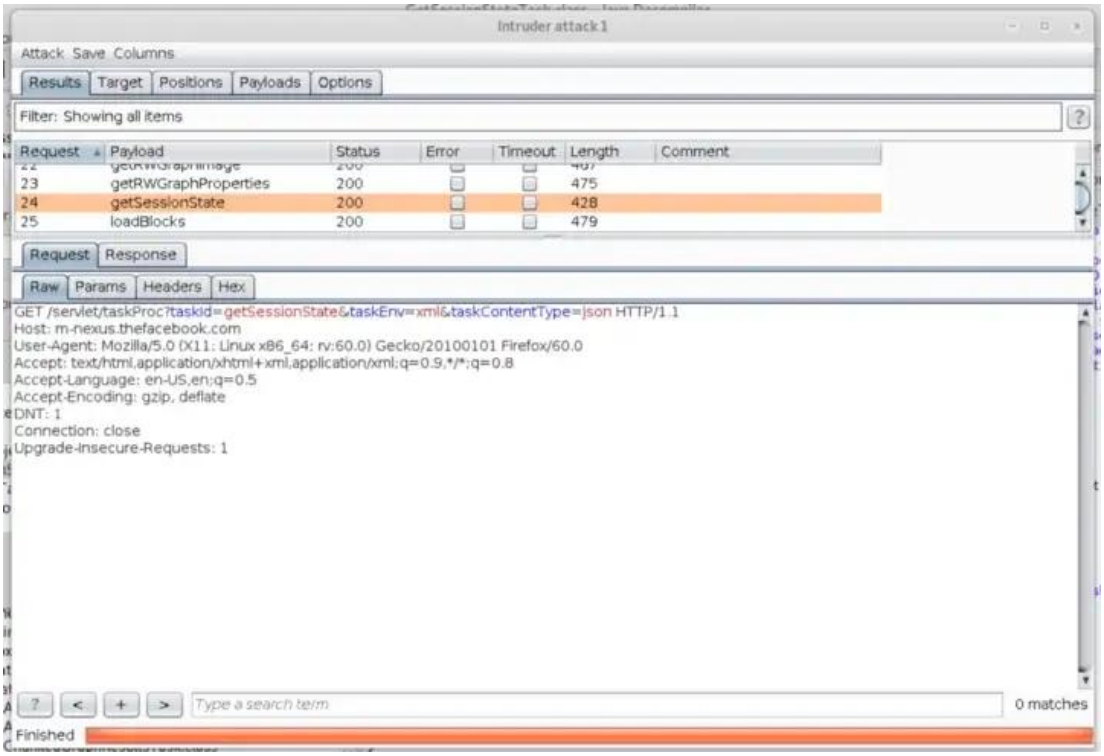


From the official configuration document for MicroStrategy, I found there are two endpoints which are publicly accessible:

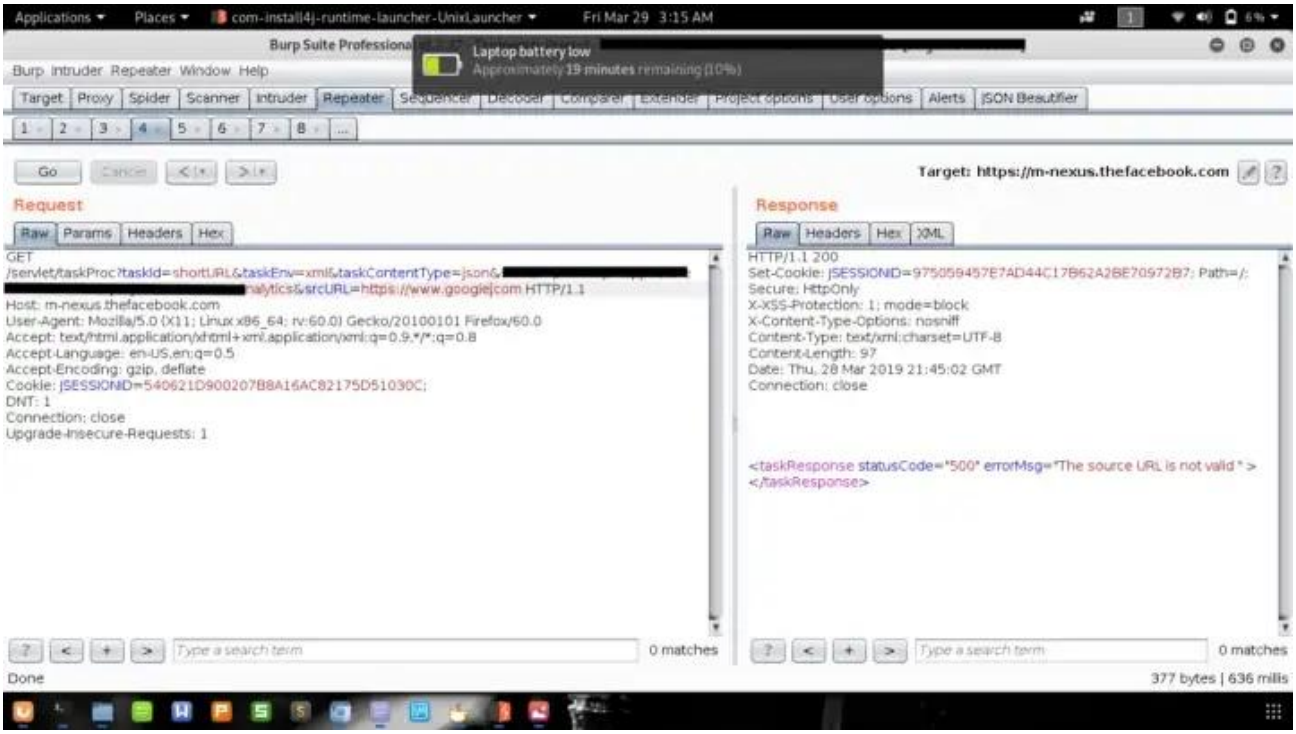


Going further in official configuration document for MicroStrategy, I found that by default, HTTP basic authentication was enabled on Business Intelligence Portal (URL: "https://m-nexus.thefacebook.com/servlet/mstrWeb"), then I observed that "https://m-nexus.thefacebook.com/servlet/taskProc" does not require authentication.

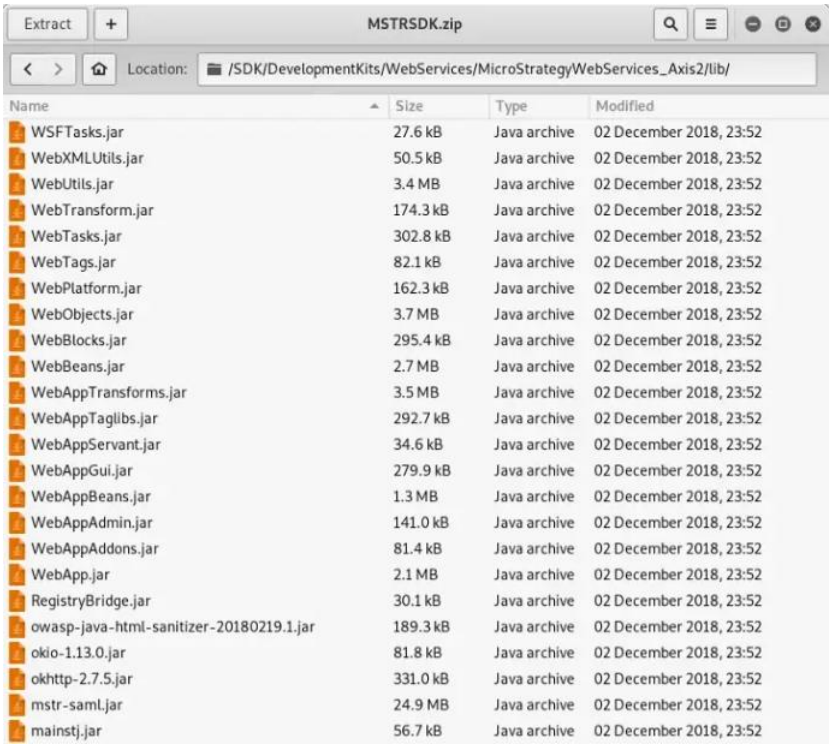
It takes value from "**taskId**" parameter to perform some custom data collection and content generation. By enumerating pre-built tasks (Using Intruder), I found that each pre-built task checks for a valid authentication session parameter, but "**shortURL**" task which processes short URL and does not check for a valid authentication session. An attacker can use this observation to access this service without any authentication.



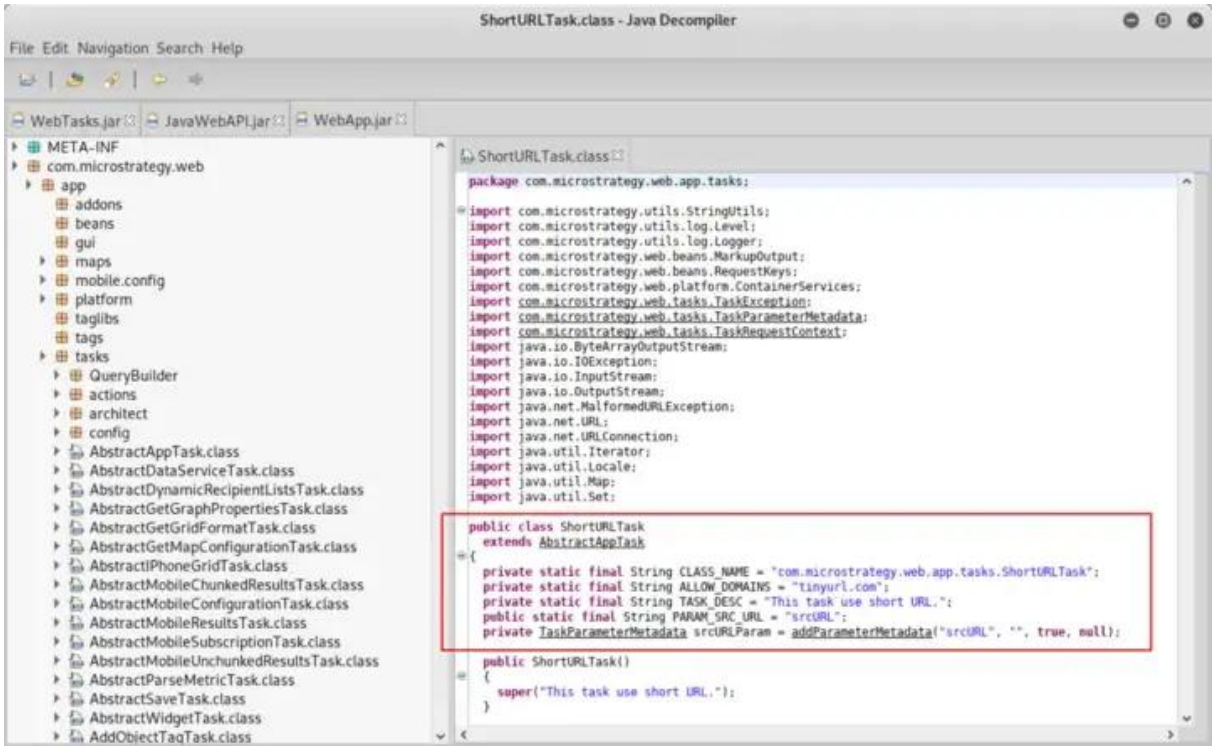
Pre-built tasks enumeration using Burp suite intruder



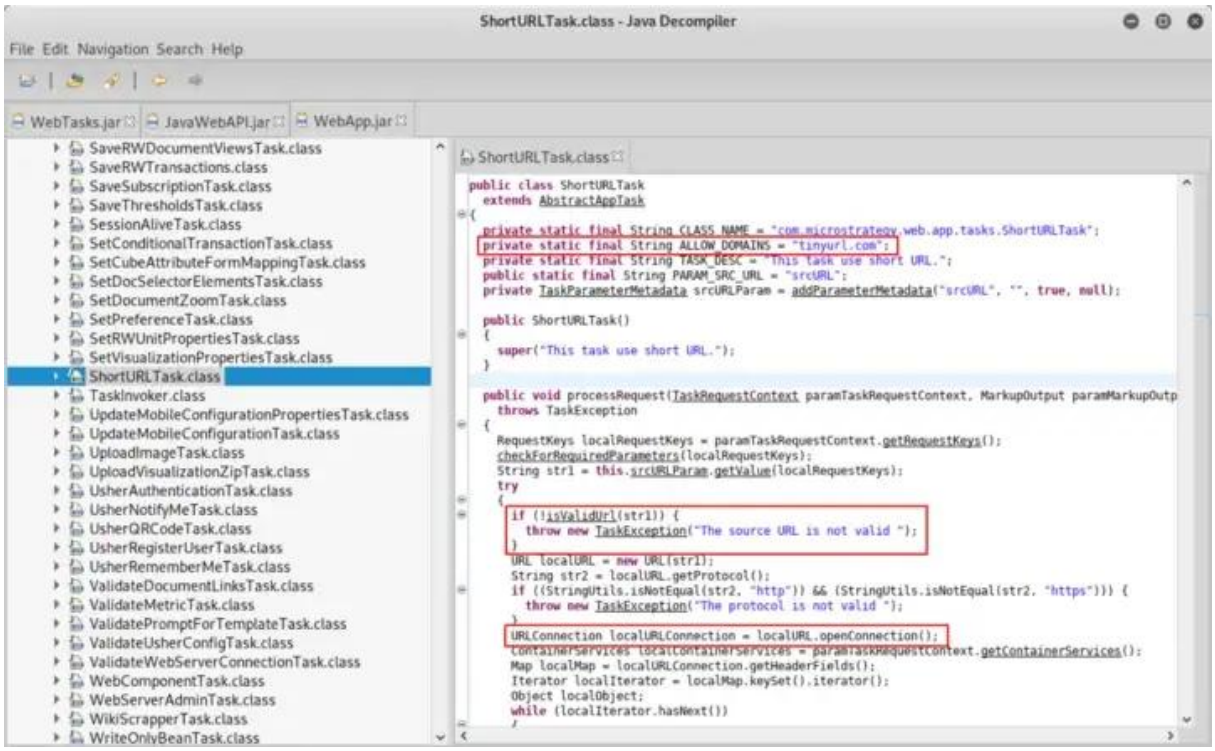
I started fuzzing on all the parameters mentioned in official document but I didn't find anything. 😔 Every time it gives me an error message "*The source URL is not valid*" with status code 500. Then I thought, let's download the hosted web application and start the source code review. I downloaded an application package of more than 400 MB. There were several scripts and jar files in the package.



Simply I decompiled that jar files using **jd-gui** tool and started reviewing code. My main target was **shortURL** task which processes short URL and does not check for a valid authentication session. Finally I found that Java class from a jar file.



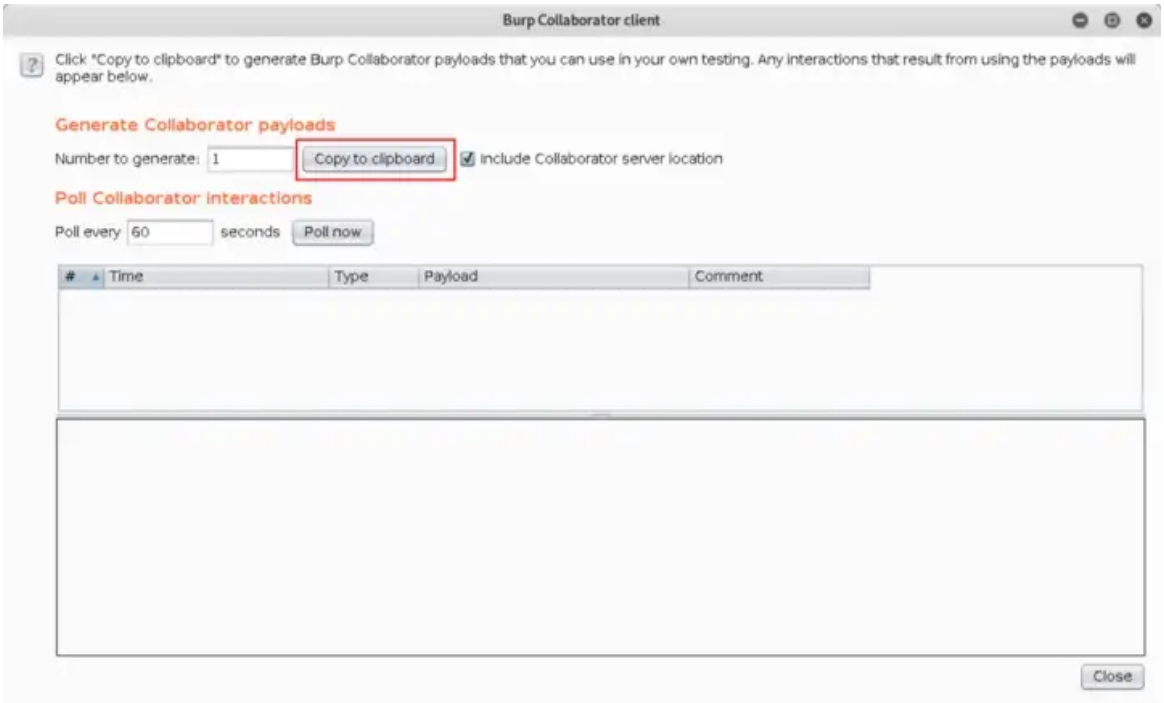
Then I came to know why it gives the same error message every time, "**srcURL**" parameter of the "**shortURL**" task only takes the URL that is created with "**https://tinyurl.com/**" for importing data or reading data from that URL. Observe the following code snippet:



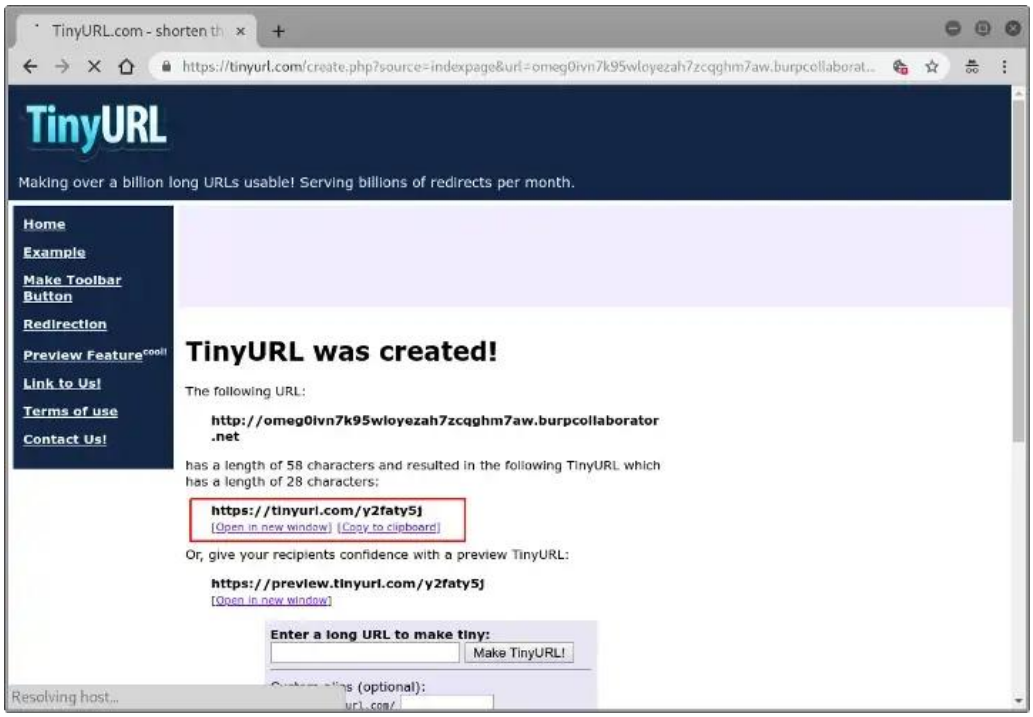
Now what? — Let's Exploit! 💥

Steps to replicate (What I sent to Facebook):

1. Open Burp suite proxy tool and go to the Burp menu and select "Burp Collaborator client". Generate a Collaborator payload and copy this to the clipboard.



2. Open "https://tinyurl.com/" from web browser, enter collaborator payload and create its tiny URL. Copy that created tiny URL.



3. Insert copied “tiny URL” in “srcURL” parameter of following URL and opens it in browser:
https://m-nexus.thefacebook.com/servlet/taskProc?taskId=shortURL&taskEnv=xml&taskContentType=json&srcURL={YOUR_TINY_URL_HERE}

4. Observe that Burp Collaborator hits immediately, it shows IP address “**199.201.64.1**” from where request was received.

This shows external **SSRF** vulnerability is present.

5. IP — 199.201.64.1 belongs to Facebook, confirmed with whois records.

6. To test internal SSRF: create tiny URL of invalid internal IP address (eg. 123.10.123.10), insert it in “srcURL” parameter and observe there is no response from server.

7. Again create tiny URL of valid internal IP address (127.0.0.1:8080), insert it in “srcURL” parameter and observe it ask for HTTP basic authentication.

With this observation we can enumerate the internal infrastructure behind a firewalled environment. I quickly reported my findings to Facebook, but this was rejected as they didn't believe it to be a security vulnerability. Observe below response:

🤔

So what's the next? — Digging Deeper ⛏️

I had to come up with the evidence. I tried to read the internal information using URL schemas such as file://, dict://, ftp://, gopher:// etc. Also tried to fetch meta-data of cloud instances but no success.

After some time I finally came up with some impactful examples. Here are some real-time attack scenarios which I sent to Facebook along with the steps to reproduce:

1. Reflected Cross Site Scripting (XSS):

2. Phishing attack with the help of SSRF:

STEP 1. Create and host a phishing page of Facebook login that steal victims Facebook login credentials which look like a legitimate login portal.

I hosted it on a private server i.e http://ahmedabadexpress.co.in/

STEP 2. Open "https://tinyurl.com/" from web browser, create a tiny URL of hosted phishing page i.e. "http://ahmedabadexpress.co.in/fb/login/fb.html". Copy that created tiny URL.

STEP 3. Insert copied “tiny URL” in “srcURL” parameter of following URL and send to victim:
https://m-nexus.thefacebook.com/servlet/taskProc?taskId=shortURL&taskEnv=xml&taskContentType=json&srcURL={YOUR_TINY_URL_HERE}

As soon as victim enters his/her username and password on this page it gets saved to the “http://ahmedabadexpress.co.in/fb/login/usernames.txt" file. And victim gets redirected to real Facebook login page. You can see that host name is the string “m-nexus.thefacebook.com” so it looks legitimate.

STEP 4. Navigate to “http://ahmedabadexpress.co.in/fb/login/usernames.txt" URL and observe stolen credentials.

An attacker might also use this vulnerability to redirect users to other malicious web page that are used for serve malware and similar attacks.

3. Fingerprint internal (non-Internet exposed) network aware services:

I was able to scan the internal network behind the firewall. I used burp suite intruder to send more than 10000 requests to find an open port on the server or any application running on that port.

After scanning, I finally found an application running on port 10303 named as "**LightRay**".

Before I further investigate on this, Facebook Security Team resolved that vulnerability.

And finally:

... is this the end? — No, the story has just begun 😊

Now I knew that the MicroStrategy Web SDK was hosted on Facebook's production server. The MicroStrategy Web SDK is written in the Java programming language and I love finding bugs in the java code. So, I decompiled each jar file of the SDK using the **JD Decompiler** tool and started reviewing the code. I've also hosted the SDK on my server, so that if I find anything suspicious in the code, I can check it there. 🕵️

After 26 days of diligence and grinding, I finally got an interesting observation. 💡

In "com.microstrategy.web.app.task.WikiScrapperTask" class, I observed that the string "str1" is initialized by user supplied input that we are sending as parameter. It will check if provided string starts with http:// (or https://) or not, if it is then it will call the function "webScrapper".

“webScrapper” function will internally send a GET request to provided URL using JSOUP. It used to fetch and parse an HTML page.

BOOM!! Again it was SSRF.. 🕺

Unfortunately, this time it was a blind SSRF so I cannot prove that it allows the submission of internal GET requests. However, from the source code of the MicroStrategy Web SDK (which is deployed on the domain m-nexus.thefacebook.com) I confirmed that it was an internal SSRF.

From this observation we cannot enumerate the internal infrastructure behind a firewalled environment or are not able to leak any sensitive information. I knew if I report this to Facebook they will reject it because there is no impact of this vulnerability. 😔

So what's the next? — I was blank at this moment!

I left it and started looking for a new vulnerability in the main Facebook domain (facebook.com)

A few months later.. ⌛

I found another vulnerability on Facebook, in which URL shortener could leak sensitive information about the server.

> *URL shortening is a technique on the World Wide Web in which a Uniform Resource Locator may be made substantially shorter and still direct to the required page. - Wikipedia*

Facebook has its own URL shortening service on **https://fb.me/**. This URL shortening service is used by both internal (Facebook employees) and public users. I notice that the short URL will simply redirect users to the long URL using HTTP Location header. I observed that the site "fb.me" did not have rate-limit set. I was looking for existing (and/or hidden) web directories and files. I launched a **dictionary based brute force attack** (around 2000 words) against that web server and analyzed the response. With the help of burp suite intruder, I captured several short links, which redirect the user to the internal system, but the internal system will redirect the user to the main Facebook domain (i.e. facebook.com).

here is a scenario:

https://fb.me/xyz ==> 301 Moved Permanently — https://our.intern.facebook.com/intern/{some-internal-data} ==> 302 Found — https://www.facebook.com/intern/{some-internal-data} ==> 404 Not Found

Note here, that some short links redirecting the user to the internal system are generated by Facebook's internal staff. It may contain sensitive internal information. such as "https://our.intern.facebook.com/intern/webmanager?domain=xyz.com&user=admin&**token=YXV0aGVudGljYXRpb24gdG9rZW4g**"

Observe HTTP response of **https://fb.me/err** URL in burp suite proxy tool which shows internal full path of logs folder.

I have been able to get more information like this using a word list and intruder. I have created a simple python script to automate this task.

Observe below screenshots of the information I got during the testing.

I have added only two screenshots. Due to Facebook policy, I cannot disclose all information. This vulnerability discloses internal HTTP GET query. This vulnerability discloses the information about the internal path to the logs folder, other file paths, internal system queries that use fetch data, internal IP address, internal ID, configuration related information, private documents etc without any authentication. By exploiting this vulnerability, it would be possible for an attacker to enumerate valid internal URLs present in the system.

Vulnerability chaining

Now I have two vulnerabilities:

1. Blind SSRF — Submit GET requests to internal and external systems
2. Server sensitive information leakage — internal path to the logs folder, other file paths, internal system queries that use to fetch data, internal IP address, internal ID

I created a scenario that shows how the sensitive information leakage may be useful for launching specific attacks like path traversal and Server Side Request Forgery (SSRF). If an attacker is able to learn the internal IP addresses of the network, it is much easier for him/her to target systems in the internal network.

I submitted both PoCs to Facebook, and I received a reply:

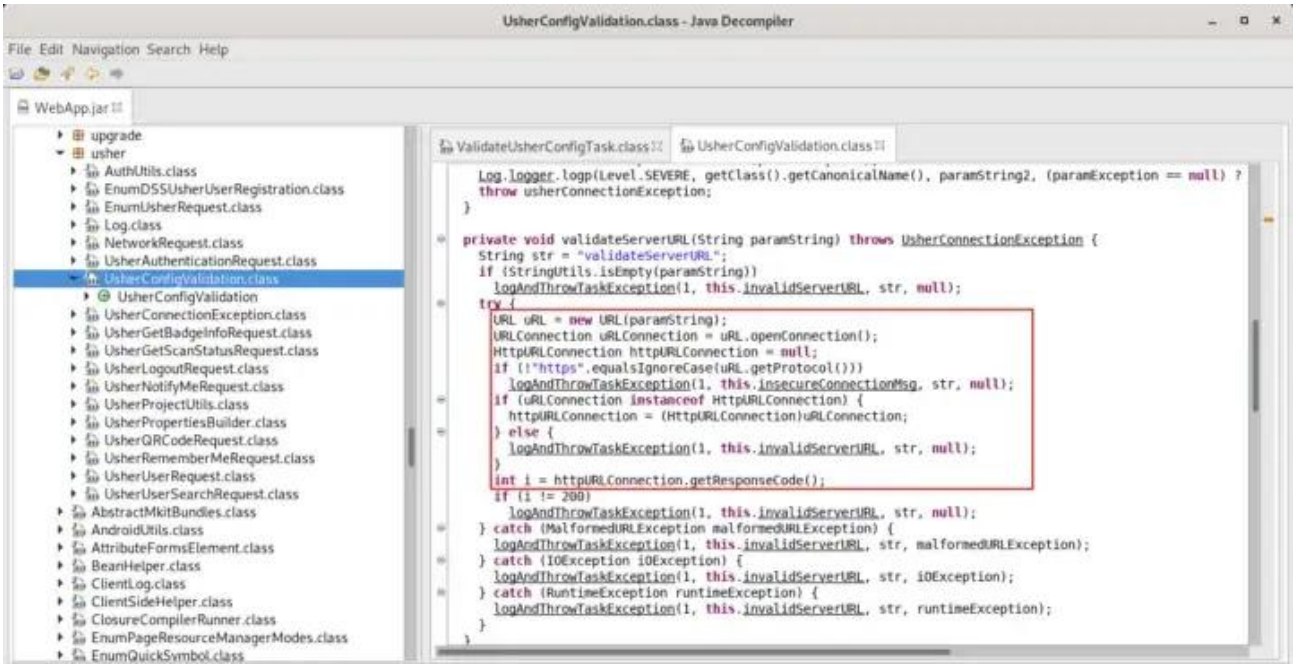
I observed that the blind SSRF bug now has been patched (“wikiScrapper” task is no more accessible/register).
Umm.. that’s not fair 😳

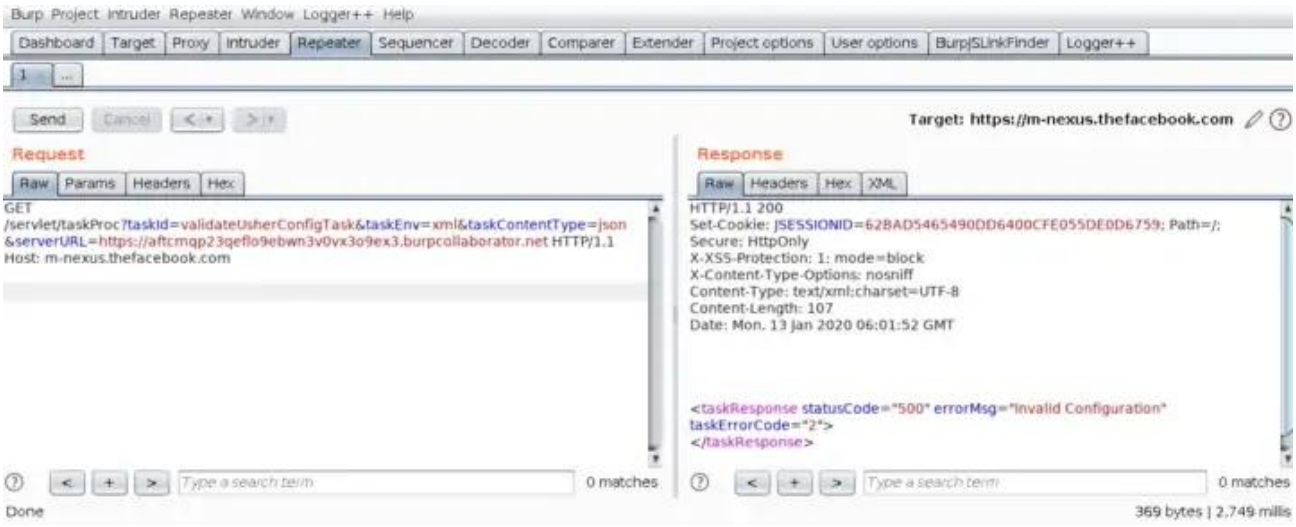
I responded:

I received a reply:

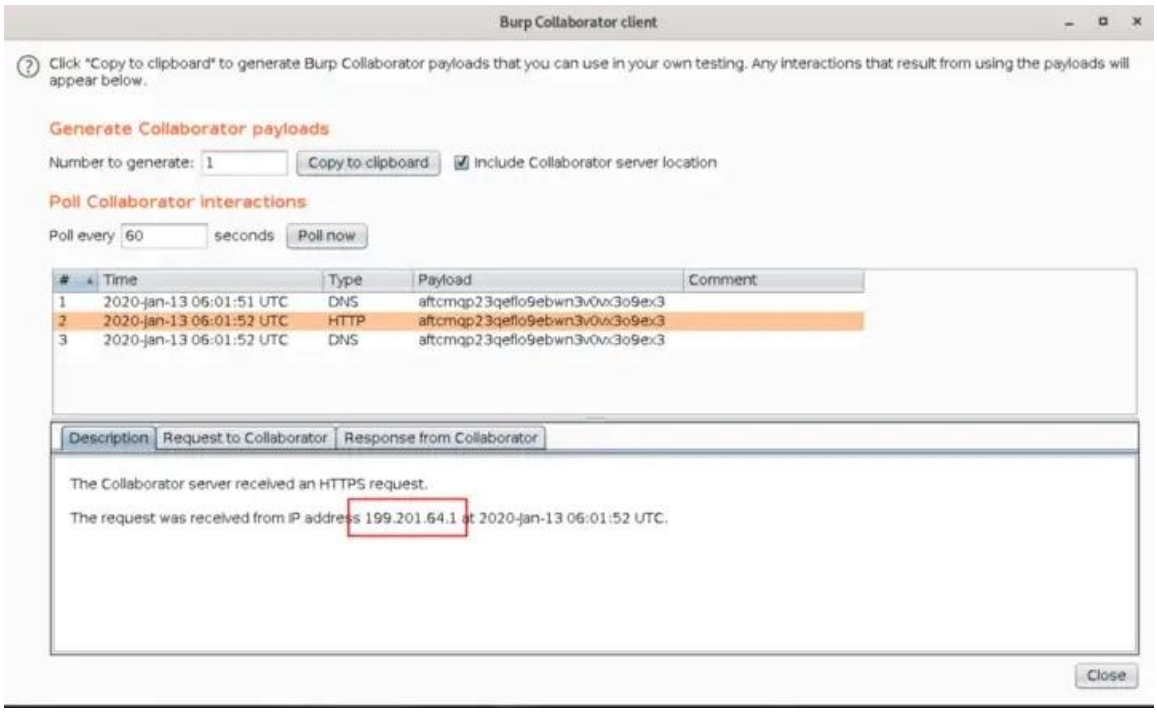
hard luck 😳

After a few days of research, I found **another blind SSRF**. 😎
From the source code of the MicroStrategy web SDK I confirmed that it is an internal SSRF. In “com.microstrategy.web.app.utils.usher” class, I observed the “validateServerURL” function which process “serverURL” parameter. The “validateServerURL” function will internally send a GET request to provided URL.





I replaced the value of "serverURL" parameter with the Burp Collaborator URL and send the GET request.



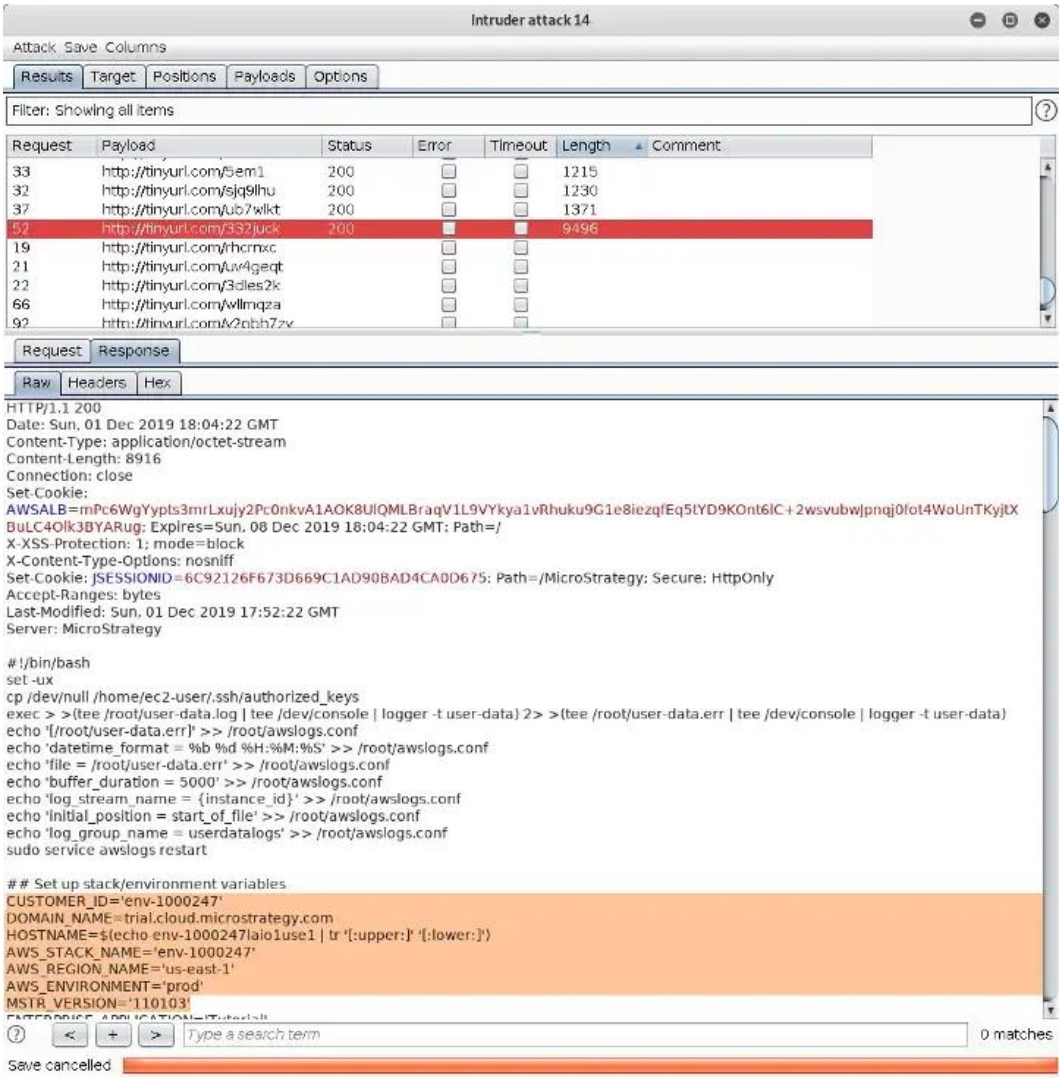
It observed that Burp Collaborator hits immediately, it will show IP address "**199.201.64.1**" from where request was received. This shows SSRF vulnerability is present.

I asked them to allow me to take the action described in my previous email.
They replied that, they are able to reproduce the bug and they are working on the patch. they will update me soon about the bounty decision.
Finally, I got a response after a few days:

Rewarded by **Facebook**

Wow! 😍😍😍😍😍

Next, I wanted to test if the SSRF vulnerability exists on the MicroStrategy demo portal, I found that this vulnerability was here as well. I was able to get some juicy information from their server using the AWS metadata API.



Response of **http://169.254.169.254/latest/user-data** meta-data API call using **SSRF**

I reported it to MicroStrategy's security team, I received the following response:

Acknowledged and rewarded by **MicroStrategy**

Conclusion

The issue has now been fixed. This is a slightly longer article, but my intention was to give you a very good understanding of how you can combine all your skills such as secure code review, enumeration and scripting knowledge to find a critical vulnerability.

When I first got this bug on Facebook server I tried to convert it to RCE but unfortunately they implemented good security measures. However, I made a total of **$31500** ($1,000 + $30,000 + $500) from this vulnerability. 💰

I hope you enjoyed the article. Pardon me for my mistakes.
Thanks for reading. Keep learning.
Stay safe and healthy 😇

Security Infosec Bug Bounty Hacking Programming