



DEPARTMENT OF COMPUTER SCIENCE

TDT4237 SOFTWARE SECURITY AND DATA PRIVACY

Exercise 3. Mitigating Vulnerabilities

GROUP 19

Author(s):
Andrea Ritossa
Nerea Francés Pérez
Oskar Hetey

Table of Contents

1	Introduction	1
2	WSTG-ATHN-01 / WSTG-CRYP-03 Sensitive Information Sent via Unencrypted Channels(TLS)	2
3	WSTG-ATHN-03 Unlimited login attempts, no logout	3
4	WSTG-SESS-01 No timeout on email verification	5
5	WSTG-SESS-01 Weak email verification link.	8
6	WSTG-ATHZ-02 Approve certification as a normal user	9
7	WSTG-SESS-06 Access token only deleted client side	10
8	WSTG-INPV-02 WSTG-CLNT-03 WSTG-CLNT-05 Unsanitized html field allowing injection.	11
9	WSTG-INPV-05 SQL injection when finishing a help request	12
10	WSTG-CRYP-04 Insecure password hasher	14
11	WSTG-CONF-12 CSP Default source not set	15
12	Conclusion	16

1 Introduction

This report has been prepared by three second-year Computer Engineering students from Spain, Germany and Italy for the course TDT4237. The overall task of this exercise is to mitigate a predefined set of 10 vulnerabilities found within the application SecureHelp, that was provided by our teachers. In order to mitigate these vulnerabilities, we first defined a Mitigation Strategy for each vulnerability and then implemented it in the code.

Each vulnerability refers to a specified section of OWASP WSTG. For a deeper understanding of this report, it is suggested to review the OWASP page. In this report, we explain how we have mitigated each vulnerability and show the code differences: all code references can be found in our GitHub repository [1]. The initial code provided is in the main branch and our implementation can be found in the production branch.

2 WSTG-ATHN-01 / WSTG-CRYP-03 Sensitive Information Sent via Unencrypted Channels(TLS)

Sensitive information sent through Unencrypted Channels (TLS) makes the website vulnerable as it can expose data to unauthorized accesses.

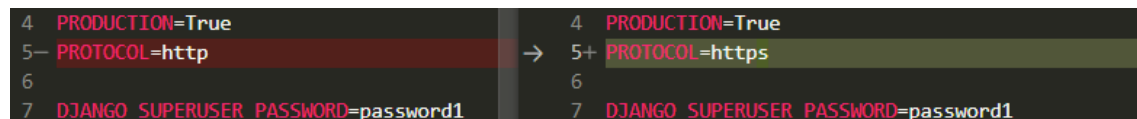
For example, if a web application sends authentication credentials or JSON Web Tokens (JWTs) over an unencrypted channel, an attacker can capture them and use them to impersonate the user or access their account. [2] [WSTG-SESS-10]

2.1 Mitigation strategy

The mitigation is to use encrypted channels (HTTPS in our case) for all communications that involve sensitive data. [2] [WSTG-CRYPT-03]

2.2 Code change

Change the PROTOCOL .env variable to https:

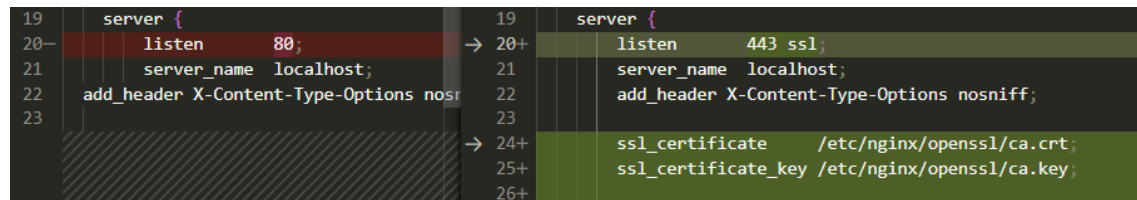
A side-by-side comparison of a .env file. The left side shows the original state with line 5 as '5- PROTOCOL=http'. The right side shows the modified state with line 5 as '5+ PROTOCOL=https'. Other lines remain unchanged: line 4 is '4 PRODUCTION=True' and line 7 is '7 DJANGO_SUPERUSER_PASSWORD=password1'.

```
4 PRODUCTION=True
5- PROTOCOL=http
6
7 DJANGO_SUPERUSER_PASSWORD=password1
```

```
4 PRODUCTION=True
5+ PROTOCOL=https
6
7 DJANGO_SUPERUSER_PASSWORD=password1
```

Figure 1: .env file

Change the port to 443 to support encrypted connection using SSL/TLS, with a change in docker-compose.yml as well. And specify the *self signed* certificate and key, after creating them with openssl.

A side-by-side comparison of the nginx.conf file. The left side shows the original configuration with line 20 as '20- listen 80;'. The right side shows the modified configuration with line 20 as '20+ listen 443 ssl;'. Additionally, lines 24-26 on the right side show the addition of SSL certificate and key paths: '24+ ssl_certificate /etc/nginx/openssl/ca.crt;', '25+ ssl_certificate_key /etc/nginx/openssl/ca.key;', and '26+'.

```
19 server {
20- listen 80;
21     server_name localhost;
22     add_header X-Content-Type-Options nosniff;
23 }
→ 24+ ssl_certificate /etc/nginx/openssl/ca.crt;
25+ ssl_certificate_key /etc/nginx/openssl/ca.key;
26+
```

Figure 2: nginx.conf file

3 WSTG-ATHN-03 Unlimited login attempts, no logout

WSTG-ATHN-03 deals with testing for weak logout mechanisms. Account logout mechanisms are used to mitigate brute force attacks: an example of the attacks that can be defeated by using a logout mechanism include login password or username guessing attack.

3.1 Mitigation strategy

Explain how you proceed to mitigate the vulnerability with a high level of abstraction. The first step is to get a deeper understanding of how the login was handled between frontend and backend: which functions are involved.

The function that from the backend operates on the login request is in views.py, line 70:

```
class LoginViewSet(viewsets.ModelViewSet, TokenObtainPairView):
    """ViewSet for logging in users. Extended from TokenObtainPairView"""
    serializer_class = LoginSerializer
    permission_classes = (AllowAny,)
    http_method_names = ['post'] # Only allow POST requests

    def create(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)

        try:
            serializer.is_valid(raise_exception=True)
        except TokenError as e:
            raise InvalidToken(e.args[0])

        return Response(serializer.validated_data, status=status.HTTP_200_OK)
```

The application is missing a layer that checks the behaviour of the user before accessing the authentication made with simpleJWT.

There are multiple possibilities to mitigate it and we opted for implementing ratelimit from django_ratelimit [3].

3.2 Code change

We added the @ratelimit decorator to the create method and set the rate limit to 10 requests per minute based on the user's IP address. If a user exceeds this limit, their request will be blocked.

```
# /views.py
from django.utils.decorators import method_decorator
from django_ratelimit.decorators import ratelimit
#[...]

@method_decorator(ratelimit(key='ip', rate='10/70m'), name='create')
class RegistrationViewSet(viewsets.ModelViewSet, TokenObtainPairView):
    """ViewSet for registering new users"""
    serializer_class = RegisterSerializer
    permission_classes = (AllowAny,)
    http_method_names = ['post']

    def create(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)

        serializer.is_valid(raise_exception=True)
        user = serializer.save()
```

```

# Create refresh token for user using simplejwt
refresh = RefreshToken.for_user(user)
res = {
    "refresh": str(refresh),
    "access": str(refresh.access_token),
}

return Response({
    "user": serializer.data,
    "refresh": res["refresh"],
    "token": res["access"]
}, status=status.HTTP_201_CREATED)

```

As a result when a potential attacker exceeds 7 requests in 15 minutes has forbidden access:

```

1 backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:50 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
2 backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:52 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
3 backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:53 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
4 backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:54 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
5 backend_group_190 | Unauthorized: /api/login/
  backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:55 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
6 backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:55 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
7 backend_group_190 | Unauthorized: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:57 +0000] "POST /api/login/ HTTP/1.1" 401
  o) Chrome/110.0.0.0 Safari/537.36"
*8 backend_group_190 | Forbidden: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:17:58 +0000] "POST /api/login/ HTTP/1.1" 403
  o) Chrome/110.0.0.0 Safari/537.36"
*9 backend_group_190 | Forbidden: /api/login/
  gateway_group_190 | 10.190.0.1 - - [12/Mar/2023:18:18:17 +0000] "POST /api/login/ HTTP/1.1" 403
  o) Chrome/110.0.0.0 Safari/537.36"

```

Figure 3: Lockout after 7th access

N.B. In the test number nine the username and password were right. Nevertheless in order to block a possible bruteforce attack it is required to forbid the access.

4 WSTG-SESS-01 No timeout on email verification

WSTG-SESS-01 deals with the Session Management Schema. In particular in securehelp there wasn't a timeout on email verification. This means that an attacker could use an old email verification link to gain access to a user's account.

4.1 Mitigation strategy

The first step was to understand the management of the Register function: it uses the following serializer to send the e-mail serializers.py, line 50:

```
class RegisterSerializer(UserSerializer):
    """Serializer for user registration"""
    password = serializers.CharField(
        max_length=128, min_length=1, write_only=True, required=True)
    email = serializers.CharField(
        max_length=128, min_length=1, required=True)

    class Meta:
        model = get_user_model()
        fields = ['id', 'username', 'email', 'password', 'is_volunteer']

    def create(self, validated_data):
        # [...]
        return user
```

And after the verification is handled by the VerificationView in views.py, line 100.

```
class VerificationView(generics.GenericAPIView):
    """View for verifying user registration links"""

    def get(self, request, uid):
        verified_url = settings.URL + "/verified"
        invalid_url = settings.URL + "/invalid"
        try:
            username = urlsafe_base64_decode(uid).decode()
            user = get_user_model().objects.filter(username=username).first()
            user.is_active = True # Activate user
            user.save()

            return redirect(verified_url)

        except Exception as ex:
            pass

        return redirect(invalid_url)
```

What's missing is a check on the time between the email delivery and the actual verification: fortunately the user model in 0001.initial.py, line 18, has an unused field named *date_joined*, ready to store the value of the registration.

```
migrations.CreateModel(
    name='User',
    fields=[
        ('id', models.BigAutoField(auto_created=True, primary_key=True,
            serialize=False, verbose_name='ID')),
        ('password', models.CharField(max_length=128, verbose_name='password')),
        [...]
```

```

        ('date_joined', models.DateTimeField(default=django.utils.timezone.now,
        verbose_name='date joined')),
        [...]
        ('user_permissions', models.ManyToManyField(blank=True,
        help_text='Specific permissions for this user.',
        related_name='user_set', related_query_name='user',
        to='auth.Permission', verbose_name='user permissions')),
    ],
    options={
        'verbose_name': 'user',
        'verbose_name_plural': 'users',
        'abstract': False,
    },
    managers=[
        ('objects', django.contrib.auth.models.UserManager()),
    ],
),

```

4.2 Code change

We added the `timestamp` from `django.utils` to the field `date_joined` of user:

```

class RegisterSerializer(UserSerializer):
    """Serializer for user registration"""
    password = serializers.CharField(
        max_length=128, min_length=1, write_only=True, required=True)
    email = serializers.CharField(
        max_length=128, min_length=1, required=True)

    class Meta:
        model = get_user_model()
        fields = ['id', 'username', 'email', 'password', 'is_volunteer']

    def create(self, validated_data):
        user = get_user_model().objects.create_user(**validated_data)

        user.is_active = False # set user to inactive until email is verified
        user.date_joined = timezone.now() # record when the verification email was sent
        user.save()

        # create email to send to user
        email = validated_data['email']
        email_subject = "Activate your account"
        uid = urlsafe_base64_encode(user.username.encode())
        domain = get_current_site(self.context["request"])
        link = reverse('verify-email', kwargs={"uid": uid})

        url = f"{settings.PROTOCOL}://{domain}{link}"

        mail = EmailMessage(
            email_subject,
            url,
            None,
            [email],
        )
        mail.send(fail_silently=False) # send email to user

        return user

```

Figure 4: *date_joined* declaration inside *RegisterSerializer*

And implemented the **expiration** inside the *VerificationView*:

```

104 class VerificationView(generics.GenericAPIView):
105     """View for verifying user registration links"""
106
107     def get(self, request, uid):
108         verified_url = settings.URL + "/verified"
109         invalid_url = settings.URL + "/invalid"
110+        expired_url = settings.URL + "/expired"
111
112         try:
113             username = urlsafe_base64_decode(uid).decode()
114             user = get_user_model().objects.filter(username=username).first()
115+            if user and (timezone.now() - user.date_joined) < timedelta(hours=1):
116+                user.is_active = True # Activate user
117+                user.save()
118+                return redirect(verified_url)
119+            else:
120+                return redirect(expired_url)
121+        except Exception as ex:
122+            print(ex)
123
124         return redirect(invalid_url)

```

Figure 5: Checking if the link has been accessed in less than an hour

We also added an **expiration page** in the frontend, that can be found at `"/expired"`.

5 WSTG-SESS-01 Weak email verification link.

WSTG-SESS-01 deals with the Session Management Schema. In particular in the sign-up phase the link that was sent from the server exposed the user id. The link was a base64 encoding of the username: it could easily be generated by an adversary.

5.1 Mitigation strategy

The lines of code that handle the signup are stated in the previous chapter SESS-01 No timeout on email verification.

A different way of storing the user id in authentication link is required: one way is by using a JSON Web Token (JWT) [4]. The server should first create a JWT, signed with a secret key. And in the JWT it can include the username. So when the link should be verified it can be retrieved.

5.2 Code change

The user is so encoded with JWT:

1. Header: JWT with HS256 as algorithm
2. Payload: username
3. Signature: the SECRET KEY of the server

```
69         # create email to send to user
70         email = validated_data["email"]
71         email_subject = "Activate your account"
72+
73+         # create JWT with payload: username
74+         payload = {'username': user.username}
75+         secret_key = settings.SECRET_KEY
76+         uid = jwt.encode(payload, secret_key, algorithm='HS256')
77+
78         domain = get_current_site(self.context["request"])
79         link = reverse('verify-email', kwargs={"uid": uid})
80
81         url = f"{settings.PROTOCOL}://{domain}{link}"
```

Figure 6: Class RegisterSerializer, of serializers.py

And in VerificationView, view.py line 104 we added the decoding:

```
#decode JWT and extract username from payload
secret_key = settings.SECRET_KEY
payload = jwt.decode(uid, secret_key, algorithms=['HS256'])
username = payload['username']
```

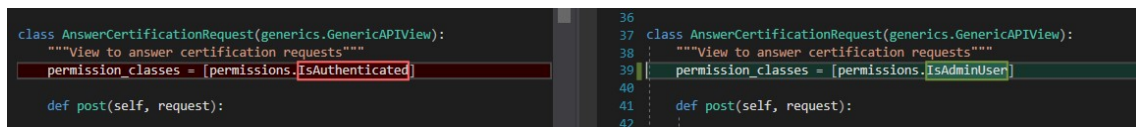
6 WSTG-ATHZ-02 Approve certification as a normal user

The *permission_classes* attribute in Django REST Framework views is used to specify the access control permissions required to access the view. Permissions are used to determine whether a user is allowed to perform a certain action on a resource.

6.1 Mitigation strategy

By changing the permission class to *permissions.IsAdminUser*, only administrators have access. We also delete *permissions.IsAuthenticated*, thus restricting access to the rest of the users.

6.2 Code change



```
class AnswerCertificationRequest(generics.GenericAPIView):
    """View to answer certification requests"""
    permission_classes = [permissions.IsAuthenticated]

    def post(self, request):

36
37 class AnswerCertificationRequest(generics.GenericAPIView):
38     """View to answer certification requests"""
39     permission_classes = [permissions.IsAdminUser]
40
41     def post(self, request):
42
```

Figure 7: updated permissions in backend/apps/certifications/views.py

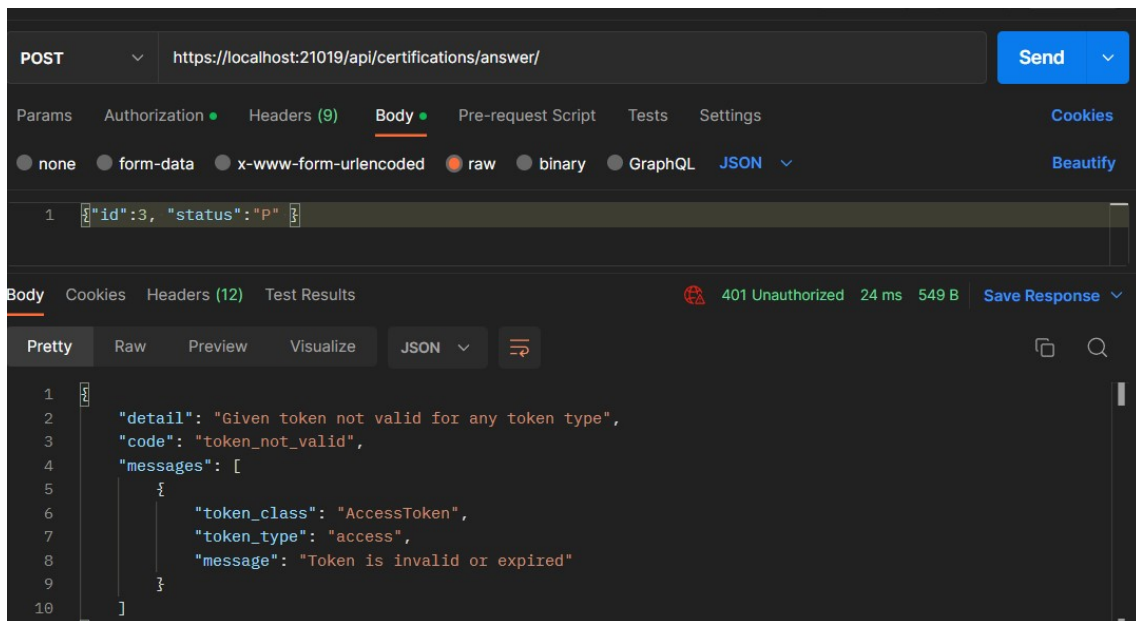


Figure 8: Postman output for approving a certification request as a normal user after changing the permissions.

7 WSTG-SESS-06 Access token only deleted client side

It is possible getting a new access token by using an old refresh token, after logging out of the application.

7.1 Mitigation strategy

To fix this vulnerability, the server-side code was updated to revoke the token upon logout, invalidating it on both the client and server-side. This was accomplished by sending a request to the server to revoke the token, along with removing it from the client-side storage. This ensures that the token cannot be used by any party after logout.

It is no longer possible to use an old refresh token to obtain a new access token. This is because the server will reject the old refresh token as invalid, and the user will have to log in again to obtain a new refresh token and access token.

7.2 Code change

```
router.register('api/login', views.LoginViewSet, basename='login')
router.register('api/refresh', views.RefreshViewSet, basename='refresh')
router.register('api/documents', views.DocumentViewSet, basename='documents')
router.register('api/logout', views.LogoutView, basename='logout')
```

Figure 9: Registering the LogoutView class in backend/apps/users/urls.py

```
class LogoutView(APIView):
    def post(self, request, format=None):
        try:
            access_token = AccessToken(request.data.get('access_token', ''))
            access_token.blacklist()
        except TokenError:
            pass
        request.user.auth_token.delete()
        return Response(status=status.HTTP_200_OK)
```

Figure 10: Invalidating tokens with the LogoutView class in backend/apps/users/views.py

```
const removeUser = () => {
  const access_token = getLocalAccessToken();
  axios.post("/api/logout/", { access_token }).then(() => {
    localStorage.removeItem("user");
    localStorage.removeItem("access_token");
    localStorage.removeItem("refresh_token");
  });
};

export { removeUser };
```

Figure 11: Sending a request to the server in frontend/src/services/token.js

8 WSTG-INPV-02 WSTG-CLNT-03 WSTG-CLNT-05 Unsanitized html field allowing injection.

To fix this vulnerability, it is necessary to properly sanitizing input data and validating it before it is displayed in HTML.

8.1 Mitigation strategy

To mitigate this vulnerability, the code was changed to use the *DOMPurify.sanitize()* function to sanitize the *description* attribute before inserting it into the HTML. This function uses a whitelist-based approach to remove any potentially malicious HTML tags or attributes from the input.

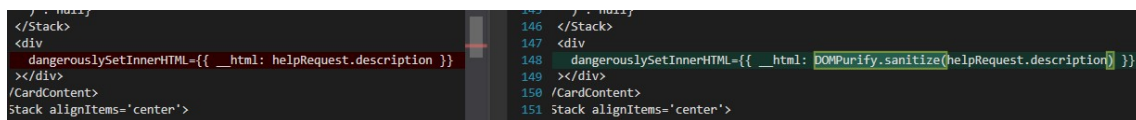
8.2 Code change



```
import HelpRequestService from "../../services/helpRequests";
import DocumentService from "../../services/documents";

11 import HelpRequestService from "../../services/helpRequests";
12 import DocumentService from "../../services/documents";
13 import DOMPurify from "dompurify";
14
```

Figure 12: import DOMPurify in frontend/src/components/HelpRequest.jsx



```
146 </Stack>
147 <div>
148   dangerouslySetInnerHTML={{ __html: helpRequest.description }}
149 ></div>
150 </CardContent>
151 <Stack alignItems='center'>
```

Figure 13: Sanitizing description attribute using DOMPurify in frontend/src/components/HelpRequest.jsx

9 WSTG-INPV-05 SQL injection when finishing a help request

The input variable for finishing help requests is not sanitized and allows for SQL injection. The basic SQL-Injection for reading out more request than it should be allowed from the solution is being used:

```
{"request_id": "1' or 1=1; --"}
```

9.1 Mitigation strategy

In this modified code, `%s` is replaced with a placeholder that is then passed to the `objects.raw()` method along with the value of `rId`. Using parameterized queries automatically escapes any special characters in the user input, making it safe to use in an SQL query.

9.2 Code change

Change the in `views.py` at query processing

```
94 def post(self, request):
95     # check if id is provided
96     if not(request.data.get('request_id')):
97         return Response({'error': 'Id is required'}, status=status.HTTP_400_BAD_REQUEST)
98
99     try: # check if id is valid
100         rId = request.data.get('request_id')
101         help_requests = HelpRequest.objects.raw(
102             "SELECT * FROM help_requests_helprequest WHERE id = %s" % rId)
103         help_request = help_requests[0]
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

Figure 14: views.py file

9.3 Comparison

These screenshots show the request results as seen in firefox before and after the code change. The different status codes and response show the results.

Status	Method	File	Type
200	POST	/api/help_requests/finish/	json
400	POST	/api/help_requests/finish/	json
200	GET	ws	
200	POST	/api/help_requests/finish/	json
400	POST	/api/help_requests/finish/	json
400	POST	/api/help_requests/finish/	json
200	POST	/api/help_requests/finish/	json

85 requests 95.65 kB / 101.60 kB transferred

Figure 15: Successful SQL-Injection

10 WSTG-CRYP-04 Insecure password hasher

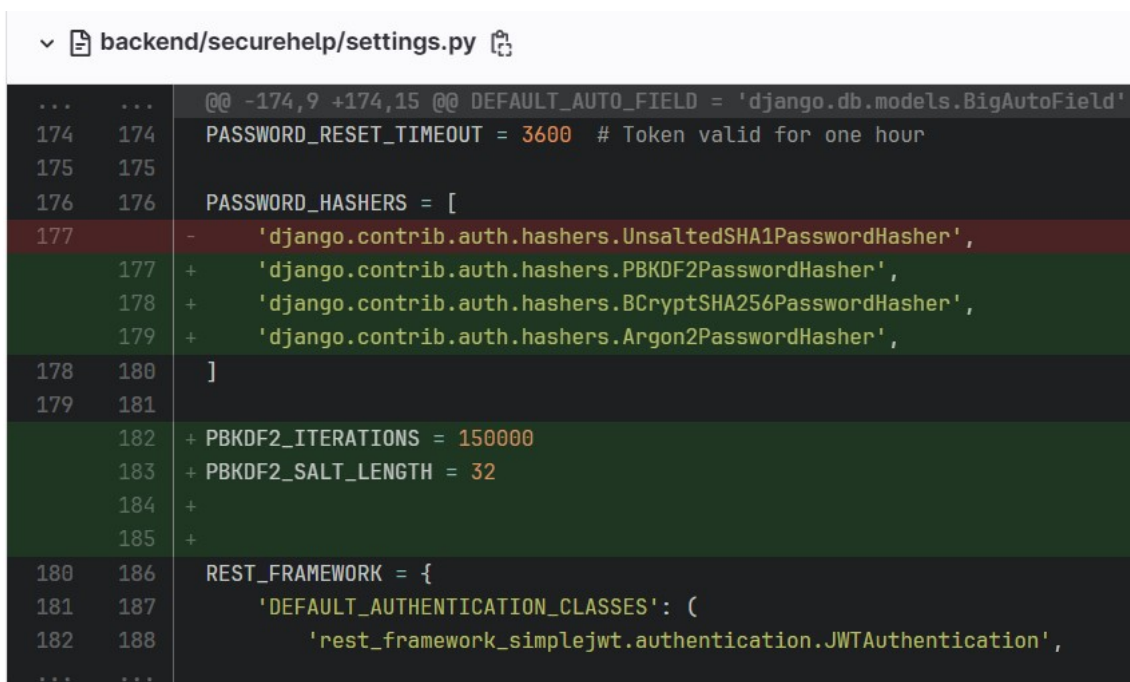
Among the outdated and insecure encryption algorithms is also found SHA1, which is being used here [5] [WSTG-CRYP-04]. Since it is being used unhashed it can simply be decoded as shown in the solutions. It is recommended to use password hashers such as PBKDF2 with a SHA256 hash function and a large number of iterations, which provides much stronger protection against password cracking attacks. [6] [Django Password Management].

10.1 Mitigation strategy

The used password hasher SHA1 is being replaced by the recommended secure algorithms. 150000 number of iterations are used to prevent attackers from attempting the cracking of passwords and also the salt length is specified so it's long enough to ensure that each salt is unique and unpredictable. Doing this in development mode results in a change of all existing passwords, so it may be needed to be still logged in as admin and create a new user after changing the algorithm to not lose access to the accounts.

10.2 Code change

The PBKDF2 password hasher was specified as the primary hasher, followed by the BCrypt and Argon2 hashers as fallbacks.



```

▼ backend/securehelp/settings.py
...    @@ -174,9 +174,15 @@ DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
174    174    PASSWORD_RESET_TIMEOUT = 3600 # Token valid for one hour
175    175
176    176    PASSWORD_HASHERS = [
177    -    'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher',
177    +    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
178    +    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
179    +    'django.contrib.auth.hashers.Argon2PasswordHasher',
178    180    ]
179    181
182    + PBKDF2_ITERATIONS = 150000
183    + PBKDF2_SALT_LENGTH = 32
184    +
185    +
180    186    REST_FRAMEWORK = {
181    187        'DEFAULT_AUTHENTICATION_CLASSES': (
182    188            'rest_framework_simplejwt.authentication.JWTAuthentication',
...    ...

```

Figure 17: updated password hashers in /backend/securehelp/settings.py

11 WSTG-CONF-12 CSP Default source not set

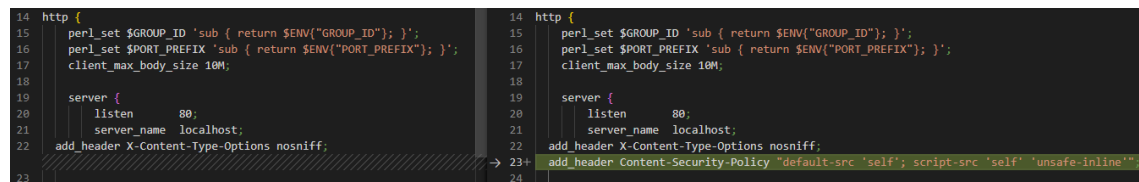
No Content Security Policy is set. Users can upload content like scripts without inspection.

11.1 Mitigation strategy

To avoid this, the nginx configuration has to be changed. An header is added in the http-block, self is being used as default source, with the aim of not blocking the original page. There have been some problems on localhost trying to figure out how to configure the header. When adding it, content may be misplaced. When changing from *self* to *localhost* even images may not be loading correctly.

11.2 Code change

The added header is found in the http section in /nginx/nginx.conf



```
14 http {
15     perl_set $GROUP_ID 'sub { return $ENV{"GROUP_ID"}; }';
16     perl_set $PORT_PREFIX 'sub { return $ENV{"PORT_PREFIX"}; }';
17     client_max_body_size 10M;
18
19     server {
20         listen      80;
21         server_name localhost;
22         add_header X-Content-Type-Options nosniff;
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Figure 18: added header in nginx.conf

12 Conclusion

Vulnerabilities in code can be exploited by attackers to gain unauthorized access to sensitive data or to disrupt the normal operation of an application. To address these vulnerabilities, developers should implement appropriate mitigation strategies that help to reduce the risk of exploitation. It is important to use secure coding practices and to perform regular security testing and code reviews in order to identify vulnerabilities in code before they can be exploited by attackers. Moreover, using security frameworks and libraries can also be an effective way to mitigate code vulnerabilities. By implementing these mitigation strategies, developers can improve the security of their applications and protect them from potential attacks.

References

- [1] group 019. *github repository of the project*. 2023. URL: <https://gitlab.stud.idi.ntnu.no/tdt4237/2023/group-019>.
- [2] OWASP. *OWASP WSTG*. URL: <https://owasp.org/www-project-web-security-testing-guide/v41/>.
- [3] django, *ratelimit*. *Django Ratelimit Docs*. 2022. URL: <https://django-ratelimit.readthedocs.io/en/stable/usage.html>.
- [4] JWT. *JWT Documentation*. URL: <https://jwt.io/>.
- [5] OWASP. *OWASP WSTG CRYPT 04*. URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/04-Testing_for_Weak_Encryption.
- [6] Django Documentation. *Django Password Management*. URL: <https://docs.djangoproject.com/en/4.1/topics/auth/passwords/>.