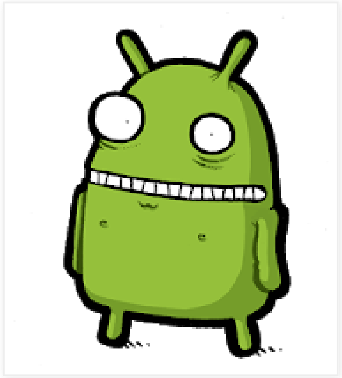
Wednesday, September 7, 2016

Android Application Security - Using hmacSHA256 Encryption For Tamper Proof Request & Response



It was all started from SSL pinning implementation. I implemented SSL pinning in our application using 3 different method as mentioned below.However I failed to implement using all 3 mechanism for obvious reasons that there are open source tools available to bypass SSL pinning.

For android there are Justtrustme, Android-SSL-TrustKiller. In iOS there is ios-SSL-Killswitch. I posted question over stackoverflow in order to find the concrete solution for the ssl pinning. However, I ended up getting nothing. Link for the stackoverflow is mentioned below.

http://security.stackexchange.com/questions/136017/is-it-possible-to-implement-secure-ssl-pinning-implementation-for-without-server

Below are the methods I tried to implement for ssl pinning.

- Using a simple HttpsURLConnection with a PinningTrustManager.
- Using a simple HttpClient with a PinningTrustManager.
- Working with PinningTrustManager and PinningSSLSocketFactory more directly.

Our end goal was simple. I did not want to give full control at client side in order to tamper with my request that can be captured within the burpsuite. For that I have implemented SSL pinning using below code.

```
// Get an instance of the Bouncy Castle KeyStore format
        KeyStore trusted = KeyStore.getInstance("BKS");
        // Get the raw resource, which contains the keystore with
```

Translate Language

Search

Search

Subscribe via email

Email address... Submit

Blog Archive

► 2020 (2)
► 2019 (6)
► 2018 (4)
► 2017 (5)
▼ 2016 (11)
  ► November 13 (1)
  ► October 30 (1)
  ► October 23 (1)
  ► October 9 (1)
  ► September 11 (1)
  ▼ September 4 (1)
    Android Application Security - Using hmacSHA256 En...
  ► July 17 (1)

```
        // your trusted certificates (root and any intermediate certs)
        InputStream   in = getResources().openRawResource(R.raw.key);
        try {
           // Initialize the keystore with the provided trusted certificates
           // Provide the password of the keystore
           trusted.load(in, KEYSTORE_PASSWORD);
        } finally {
           in.close();
        }
        String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
        TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
        tmf.init(trusted);
```

Only answer that I got from stackoverflow is mentioned below.

"No, there is no secure solution. All the mentioned tools that bypass SSL certificate pinning depending on root access.

If the phone is rooted/jailbroken there is no way for an app to enforce anything. As your application is only a piece of software running on that system, the person who owns the system (and that is the user on jailbroken/rooted devices) can always bypass your security mechanisms. The only thing you can do is to make the live of the person who wants to disable SSL certificate pinning harder. One way is to use a jailbreak/root detection and block jailbroken/rooted devices."

So I decided to think in a different way and I searched about hashing techniques in request and response. This is not PKI. However, it works as partial PKI. As I am not a developer I asked my good friends to implement that in my application. Here is the diagram how it works.

Consider below workflow:
Phase 1:

1. Application is downloaded in user's mobile which asks for few permission.
2. User installs application by allowing all asked permission.
3. Application gathers unique identifiers from the user's mobile and store on the server.

Phase 2:

1. Server computes one key using those unique identifiers which will be pertaining to that specific user only.
2. Server passes this key and stores either in keychain/shared preferences which are encrypted.

Phase 3:

1. Now client and server both have one unique key computed using user's unique mobile identifier. It can be UUID, mobile number etc.

2. Server and client both has algorithm to use with unique key in order to generate random hash.

Sample Code For Implemetation

```
public static String encode(String key, String data) throws Exception {
  Mac sha256_HMAC = Mac.getInstance("HmacSHA256");
  SecretKeySpec secret_key = new SecretKeySpec(key.getBytes("UTF-8"), "HmacSHA256");
  sha256_HMAC.init(secret_key);

  return Hex.encodeHexString(sha256_HMAC.doFinal(data.getBytes("UTF-8")));
}

public static void main(String [] args) throws Exception {
  System.out.println(encode("key", "The quick brown fox jumps over the lazy dog"));
}
```

Or you can return the hash encoded in Base64:

```
Base64.encodeBase64String(sha256_HMAC.doFinal(data.getBytes("UTF-8")));
```

The output in hex is as expected:

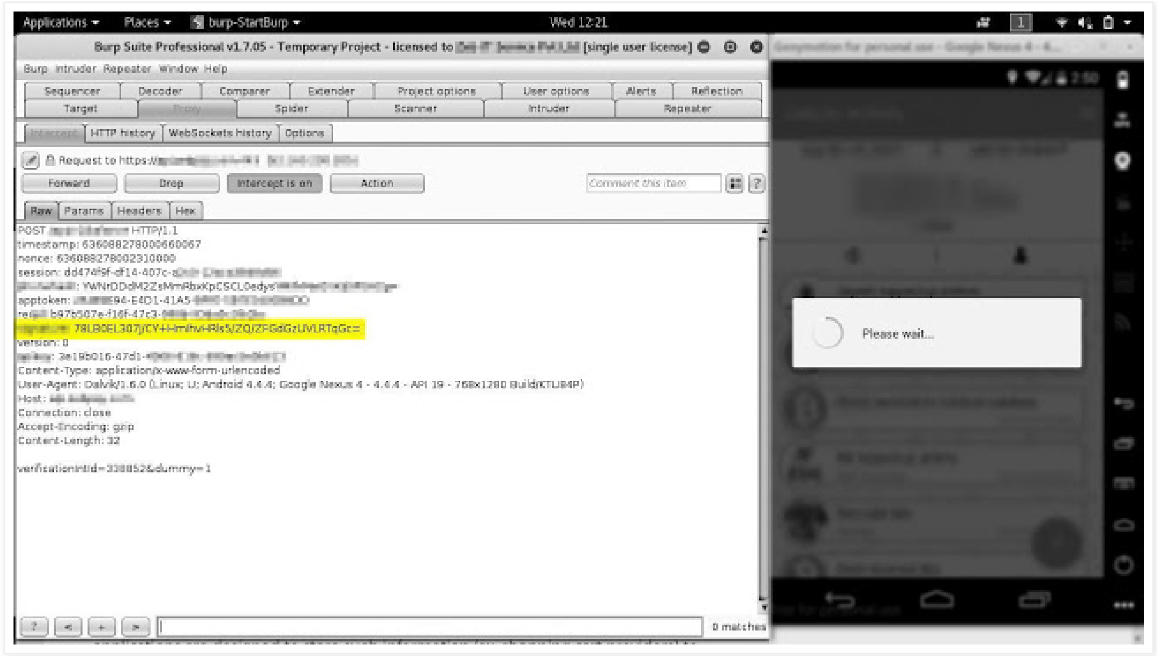
```
f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8
```

As you can determine here I am taking string and converting it into UTF-8 format as string can be anything ranging from normal alphanumeric to any Chinese or other unseen characters. And then HMACsha256 is applied on that.

Now I want to pass that calculated hash within request so I can use base64 for the same as standard practice.

How Application Actually Works:

Scenario 1: Request without tampering
Application generates one request in which there will be couple of headers and GET/POST parameters. Within the application logic must be implemented that which headers will be taken in the consideration of computing hashing string. Request will look like as follows:


Burp Suite Professional v1.7.05 - Temporary Project - licensed to [redacted] [single user license]
Forward
Drop
Intercept is on
Action
Raw
Params
Headers
Hex
Please wait...

```
Request to https://[redacted]
POST [redacted] HTTP/1.1
timestamp: 636088278000660067
nonce: 636088278002310000
session: dd474f9f-df14-407c-[redacted]
[redacted]: YWNrDDdM2ZsMmRbxKpCSCL0edys[redacted]
apptoken: [redacted]94-E4D1-41A5-[redacted]
re[redacted] b97b507e-f16f-47c3-[redacted]
[redacted] 78LB0EL307j/CY+HmIhvHRls5/ZQ/ZFGdGzUVLRTqGc=
version: 0
[redacted]: 3e19b016-47d1-[redacted]
Content-Type: application/x-www-form-urlencoded
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.4; Google Nexus 4 - 4.4.4 - API 19 - 768x1280 Build/KTU84P)
Host: [redacted]
Connection: close
Accept-Encoding: gzip
Content-Length: 32

verificationIntId=338852&dummy=1
```

As you can observe that application is in "Please wait" mode as we have not forward the request to the server so far. Highlighted yellow part is the signature generated by application using some per-defined headers, their values and post/get parameters and their values. That logic is written within the application. Now lets forward this request without any tampering.

You will observe in the below screenshot that application responded normally without any error message.

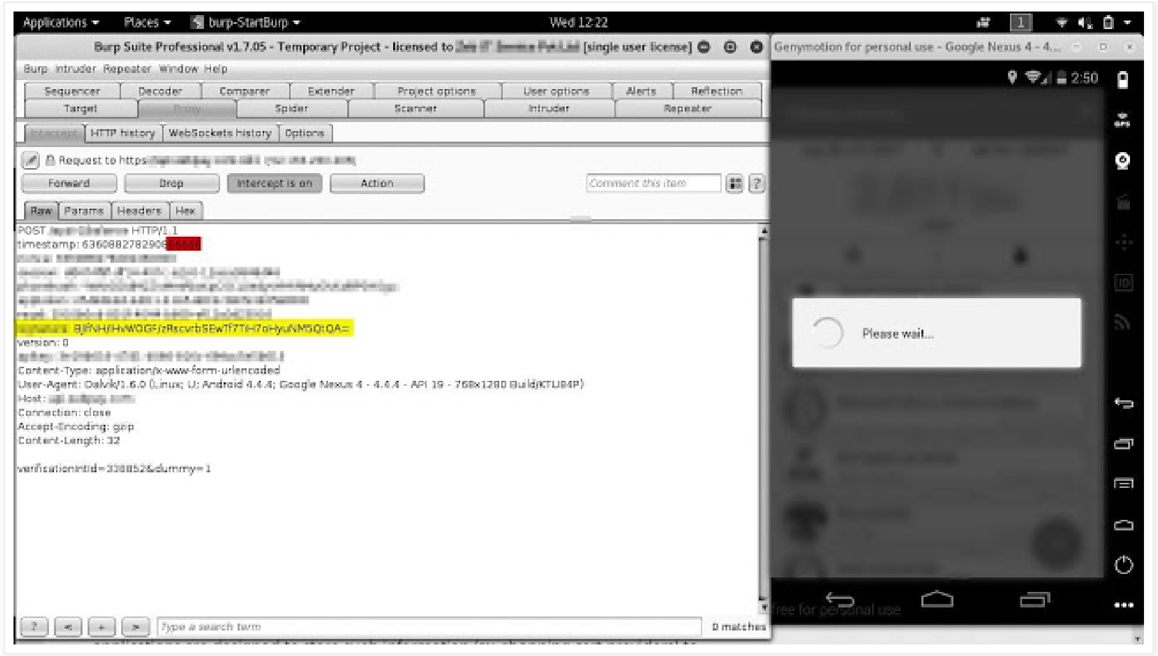
Applications Places burp-StartBurp Wed 12:21
Burp Suite Professional v1.7.05 - Temporary Project - licensed to [single user license]
Burp Intruder Repeater Window Help
Sequencer Decoder Comparer Extender Project options User options Alerts Reflection
Target Proxy Spider Scanner Intruder Repeater
Intercept HTTP history WebSockets history Options
Forward Drop Intercept is off Action
Raw Params Headers Hex
0 matches
Genymotion for personal use - Google Nexus 4 - 4...
2:50

Due to some confidential information, I have obfuscated the screenshot.

Scenario 2: Request with tampering

As highlighted below in red color, I am modifying the timestamp header's few digits and forward request to the server. Make sure as a black hat assessment, I am until now unaware about key and computational algorithms, hence I do not know that I have to change the yellow highlighted hash value too in order to make this request work on the server side.

Because I changed the digit so hash must not be same as mentioned below.

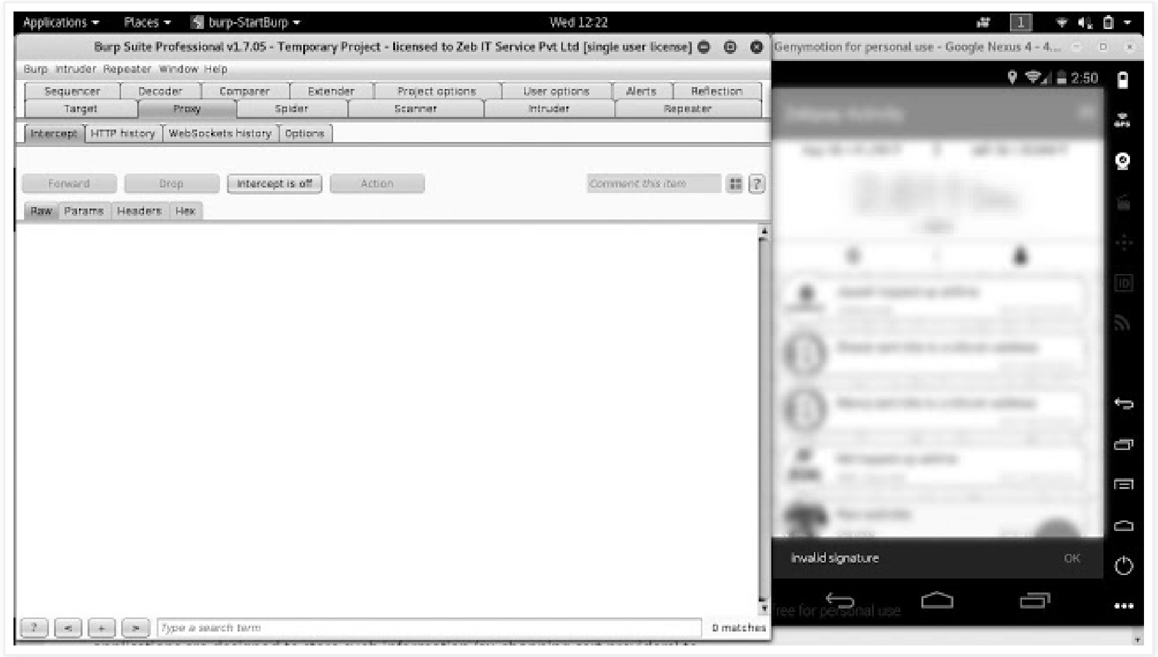


Request to https[redacted]

```
POST [redacted] HTTP/1.1
timestamp: 63608827829066666
[redacted]
[redacted] BJfNH/HvWOGF/zRscvrbSEwTf7TiH7oHyuNM5QtQA=
version: 0
[redacted]
Content-Type: application/x-www-form-urlencoded
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.4; Google Nexus 4 - 4.4.4 - API 19 - 768x1280 Build/KTU84P)
Host: [redacted]
Connection: close
Accept-Encoding: gzip
Content-Length: 32

verificationIntId=338852&dummy=1
```

Please wait...

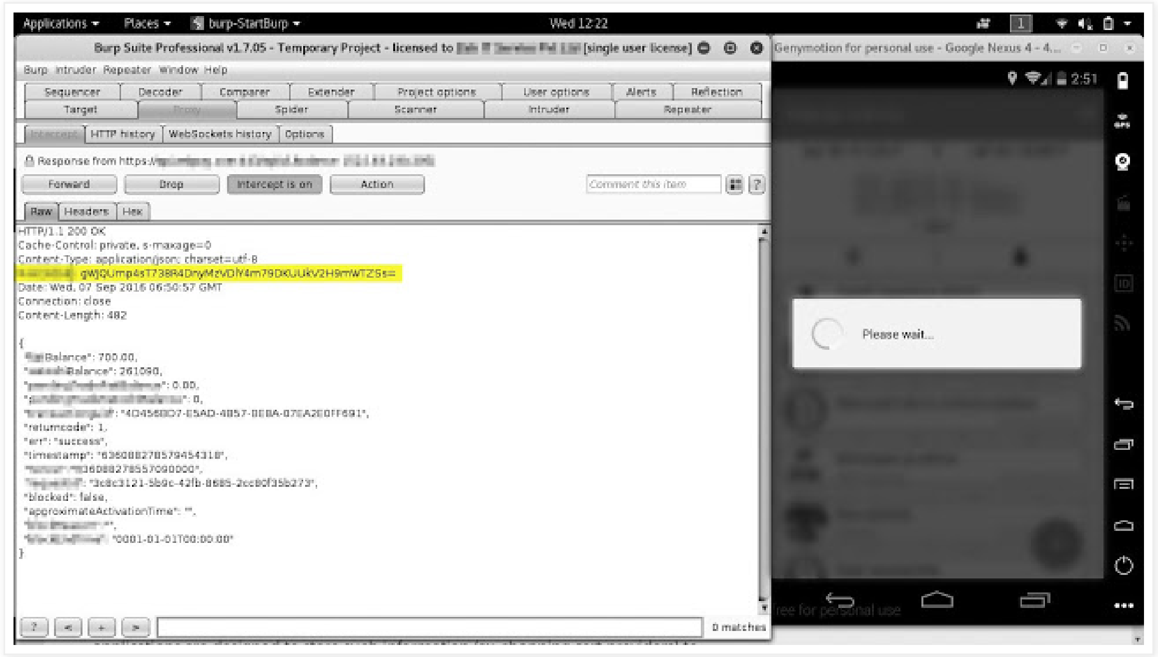
Now server receives the modified request and it also computes the hash at its end. Now that hash and original yellow highlighted hash does not match. Hence server will give error message in the application as mentioned below "invalid signature".



Invalid signature OK

Scenario 3: Response without tampering

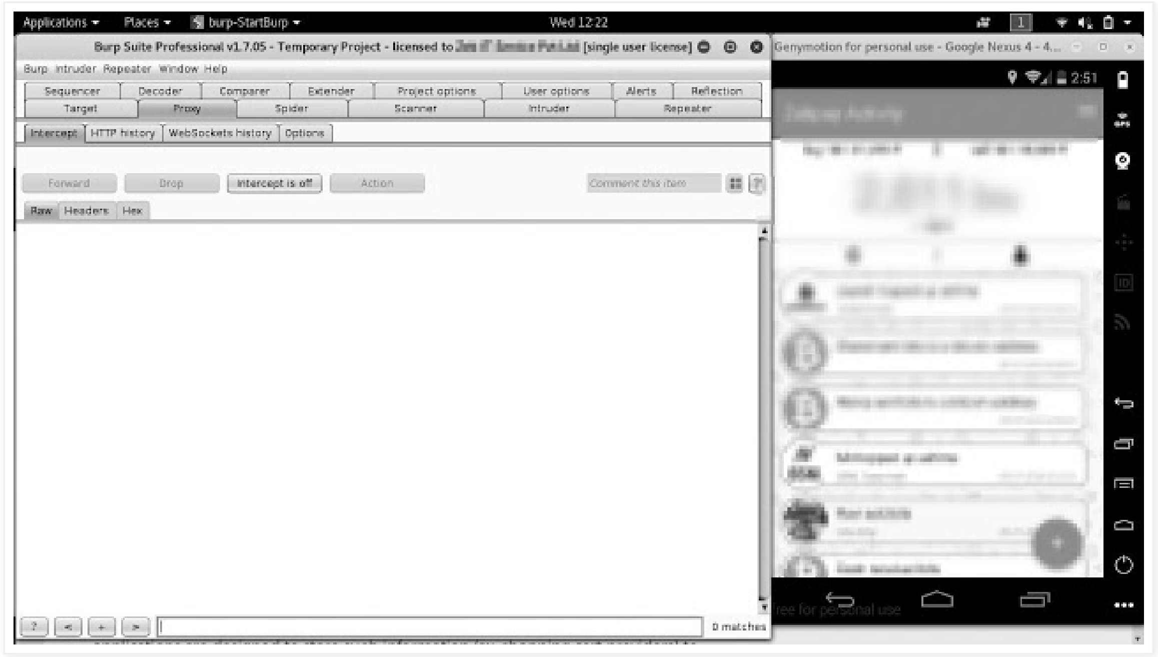
Below is the original response received in the burpsuite. Now it is in json format so server has taken whole json body response and calculated the hash accordingly.



Response from https:[redacted]

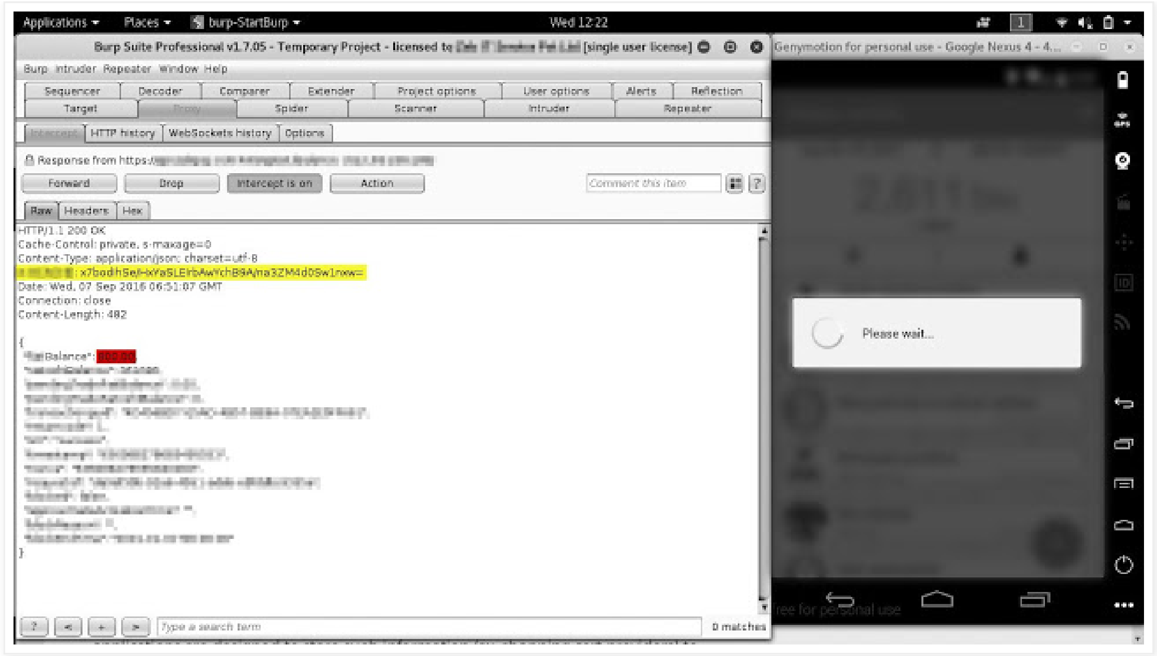
```
HTTP/1.1 200 OK
Cache-Control: private, s-maxage=0
Content-Type: application/json; charset=utf-8
[redacted] gWJQUmp4sT738R4DnyMzVDlY4m79DKUUkV2H9mWTZ5s=
Date: Wed, 07 Sep 2016 06:50:57 GMT
Connection: close
Content-Length: 482

{
 [redacted]Balance": 700.00,
 "[redacted]Balance": 261090,
 "[redacted]": 0.00,
 "[redacted]": 0,
 [redacted]: "4D4560D7-E5AD-4857-0EBA-070A2E0FF691",
 "returncode": 1,
 "err": "success",
 "timestamp": "636088278579454310",
 [redacted] "636088278557090000",
 [redacted]: "3c8c3121-5b9c-42fb-8685-2cc80f35b273",
 "blocked": false,
 "approximateActivationTime": "",
 [redacted]"",
 [redacted] "0001-01-01T00:00:00"
}
```

Below is the normal response in application as we did not tamper anything within the response.



Scenario 4: Response with tampering

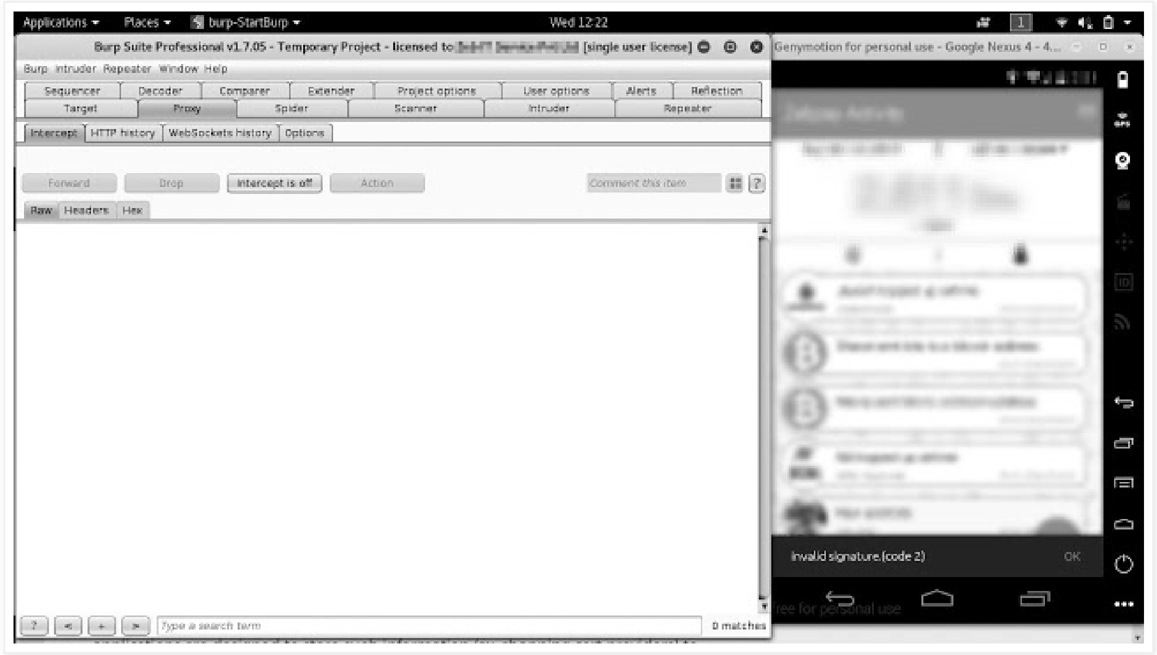
Now I am modifying the prize value from 700 to 800 as mentioned below. Note that server has calculated and sent hash according to original response which was having 700 value. Now attacker does not know what value shall he/she put according to new data so that app can computer and match.



Response from https:[redacted]

```
HTTP/1.1 200 OK
Cache-Control: private, s-maxage=0
Content-Type: application/json; charset=utf-8
[redacted]: x7bodIhSe/HxYaSLEIrbAwYchB9A/na3ZM4d0Sw1nxw=
Date: Wed, 07 Sep 2016 06:51:07 GMT
Connection: close
Content-Length: 482

{
 "[redacted]Balance": 800.00,
 [redacted]
}
```

Now application receives response and it calculates hash according to new data of 800. That hash will not match to the hash which is there in response header. Thus how application will generate error message at client side within the application.



Invalid signature.(code 2)

Thus how request and response tampering can be prevented.

Few Assumptions:

As security is just an illusion, I am making few assumptions here. I know key is stored at client side either in shared preference or in keysotre and algorithm can be also found through the source code. However, it can be possibly difficult for an attacker at some level to find both exact information and create a plugin or an extender which can generate new hash in air (runtime in burp) in order to simulate the attack. Also your source code is protected with proguard.

I know this is not a bullet proof security as PKI is not used in this. Still it provides good level of security to keep script kiddies away from your application.

Kudos to my android and iOS developer who achieved this.

References:

http://stackoverflow.com/questions/7124735/hmac-sha256-algorithm-for-signature-calculation

Posted by Frogy at 9/07/2016

Labels: android, android security, burp, burp proxy, hmac, security, sha256

No comments:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)