NASA HW11 - 金哲安(B12902118)

1. 三角準則的侵略者

References

- B12902116 (林靖昀)
- B12902066 (宋和峻)
- https://www.ithome.com.tw/news/167851

(a)

1. Bybit, a cryptocurrency exchange got hacked and had 1.46 billion US dollars worth of cryptocurrency stolen on 2025/2/21. The attacker performed phishing attack to gain access to the infrastructure of Safe{Wallet}. This violated the confidentiality principle because it was an authorized access. Moreover, the trading webpage was modified to initiate an unintended transaction, this violated the integrity principle.
2. My friend's credit card credentials got stolen and unwanted payments were made. This is a violation of confidentiality because unauthorized people gained access to credit card credentials.

(b)

- Assumption
    i. The laptop has a password lock
- Threat model:

Threat Model	**Countermeasure**
Someone tries to perform social engineering to get the password.	Be aware of suspicious people and activity and change the password when a possible compromise happens.
Someone whose permission has been revoked tries to access the laptop again.	Reset the password once someone's permission is being revoked.

(c)

- Assumption

   i. Anyone can use their phone to scan the QR code and send the text message.

- Threat model:

Threat Model	**Countermeasure**
Someone tries to tamper with another one's text message sent to the server.	Use end-to-end encryption.
Someone tries to perform DDOS attack on the server.	Monitor and reject suspicious and abnormal traffic.

(d)

- Assumption
   i. All teams are present at the classroom to take the exam.
- Threat model:

Threat Model	**Countermeasure**
Someone tries to communicate with a human from another team from a laptop.	Install monitoring software on all students' laptop.
Someone tries to communicate with a human from another team from a phone.	Monitor all students physically with TAs.

2. 果汁店也有洞！

References

- B12902116 (林靖昀)
- B12902066 (宋和峻)
- https://tech-blog.cymetrics.io/posts/jo/zerobased-cross-site-scripting/
- https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html
- https://github.com/DataDog/juice-shop/blob/master/SOLUTIONS.md
- https://chatgpt.com/share/68399a6f-8c18-8005-be3d-e92fe3bf67fb

(a)

(i) DOM XSS

Click the search icon on the header row and then search for <iframe src="javascript:alert(`xss`)">

DOM XSS is a kind of attack where the attacker inserts strings that are interpreted as DOM objects to the website and then being executed on the victim's browser.

The website rendered out my input as a DOM object and then executed it directly on the browser.

To prevent this, always validate user input and render as plain text.

(iv) Five-Star Feedback

First login as admin through SQL injection on the login page. Enter `' OR 1=1 --` for the account and `1` for the password. Then go to http://localhost:3000/#/administration. And then delete the 5 star feedback.

SQL injection is a kind of attack where user input is interpreted as a part of the SQL command to be executed in the backend.

The SQL command being executed during login is likely to be: SELECT * FROM Users WHERE email = '[input_email]' AND password = '[input_password]'. And with my input, the command becomes: SELECT * FROM Users WHERE email = '' OR 1=1 --' AND password = '1'. Anything following -- is commented out in SQL. This returns all list of users. Luckily the first user on the list is the admin, and the backend logic probably took the first row of the returned result as the user. Therefore I can login as admin.

To prevent this, always validate user input and substitute with escape characters.

(v) View Basket

On a Chrome browser, login as admin and open up Developer Tools (press F12). Go to Application > Storage > Session Storage > http://localhost:3000. Change the value of bid to 2. Then click on Your Basket on the header row.

This vulnerbility exists because the website stores user identification information as plain text in the session storage, which can be easily manipulated by an attacker.

After I have changed my bid to 2, the server will think that the request is coming from another user and therefore reply with another user's basket.

To prevent this, the user identification should be stored as a session id with at least 64 bits of entropy. This can prevent prevent brute-force session guessing attacks. The backend can use session management to map session id's to user id's.

(vi) Password Strength

After loggin with SQL injection from previous problems, open up Developer Tools (press F12). Go to Application > Storage > Cookies > http://localhost:3000. Copy the value for token and decode it with base64. We can see that there is: `"email":"admin@juice-sh.op", "password":"0192023a7bbd73250516f069df18b500"` after decoding. Go to https://crackstation.net/ and crack the password to get `admin123`. Login as admin with admin account: `admin@juice-sh.op` and password `admin123`.

This vulnerability is caused by weak password hashing. The password is hashed with md5 without adding salt. If the attacker gains access to the hash, then the attacker effectively gains access to the password.

To prevent this, use more secure hashing algorithms like sha256 or sha512.

(ix) Repetitive Registration

Logout and go to the registration page. Enter email: nasa@nasa.com, password: nasa1, repeat password: nasa1, and then change password to nasa1nasa, but don't change the repeat password. Select "Your eldest siblings middle name" as the security question and type Michael. Finally, click register.

This is caused by improper input validation done by the frontend javascript. Once the repeat password is validated as the same as the password, the frontend script doesn't re-validate if the password has changed.

To prevent this, the frontend code should fix this bug or the backend code should validate again.

(b)

SSRF (Server-Side Request Forgery) is a kind of attack where the attacker tricks a vulnerable server into making a request to an internal or external resource that the attacker controls or wants to access. The attacker will make the server make requests to services on the server's behalf, potentially giving more access.

CSRF (Cross-Site Request Forgery) is a kind of attack where the attacker tricks a user's browser into sending a request to a web application that they are already authenticated with, performing actions without their intent. The attacker might send a malicious link for the victim to click on and perform privileged actions like transactions since the victim is authenticated.

XSS (Cross-Site Scripting) is a kind of attack where the attacker injects malicious scripts into webpages viewed by other users. The attacker will let victims execute malicious commands on the victims' browsers.

The difference between the three is that SSRF makes the server to make unwanted requests, CSRF makes the victim to make unwanted requests to the server, and XSS make the victim's browser to execute unwanted commands.

3. R-SA！破密部

References

- B12902116 (林靖昀)
- B12902066 (宋和峻)
- https://en.wikipedia.org/wiki/Coppersmith's_attack
- https://en.wikipedia.org/wiki/Chinese_remainder_theorem
- https://en.wikipedia.org/wiki/RSA_cryptosystem
- https://docs.xanhacks.xyz/crypto/rsa/08-hastad-broadcast-attack/

1

Execute `code/3a.py`
Flag: `NASA_HW11{blind_signing_is_dangerous}`

For RSA, the signature of a message m is s = m^d mod n. If I have signatures for m1 and m2, I can compute the signature for m1 * m2 mod n:

```
sig(m1) = m1^d mod n
sig(m2) = m2^d mod n
sig(m1 * m2) = (m1 * m2)^d mod n = (m1^d * m2^d) mod n = sig(m1) * sig(m2) mod n
```

So I chose two messages m1 and m2 such that m1 * m2 ≡ m (mod n), where m is `b'name=soyo'`. I arbitrarily picked `m1_bytes = b'hello world'`, which doesn't start with `name=`, and computed `m2 = m * inverse(m1, n) % n`. Luckily it also doesn't start with `name=`. And then after getting both signatures, I can compute the real signature that I want by `forged_sig = (sig1 * sig2) % n`.

2

Execute `code/3b.py`
Flag: `NASA_HW11{W0w_y0u_kNow_h@st@d'5_bro4dc@s7_47t@cK}`

Extending from the previous problem, I can now also get Anon's public key (e, n) and the ciphertext c as many times as I want, but each time, a new random keypair is generated. And since the plaintext is encrypted with the same small exponent 7, I can perform Håstad's broadcast attack.

I first collect 7 pairs of (n, c), and then use the Chinese Remainder Theorem to combine the ciphertexts. `c_i = m^e mod n_i for each i` and CRT gives me `C ≡ m^e mod N where N = n_1*n_2*...*n_7`. Since `m < n_i for all i`, `m^7 < N`. Therefore `C = m^7`. I then take the 7th integer root of `C` to get the plaintext `m`.

6. 猫物語（赤）

References

- B12902116 (林靖昀)
- B12902066 (宋和峻)

(a)

Execute `code/6a.py`
Flag: `NASA_HW11{pseudorandomness_does_not_guarantee_unpredictability}`

I can recover a and c from consecutive outputs of LCG. Since I have:

```
s1 = (s0 * a + c) % m
s2 = (s1 * a + c) % m
```

I can easily solve

```
a = (s2 - s1) * inverse(s1 - s0, m) % m
c = (s1 - s0 * a) % m
```

And easily predict

```
next_number = (s2 * a + c) % m
```

So I do it 101 times in a row, and then ask for the flag.

(b)

Ask Fatcat about his exam result by entering 3 after connecting to the server. Then paste the answer sheet to `encrypted_flag_hex` in `code/6b.py`. And then execute it.
Flag: `Flag:NASA_HW11{07p_k3y_mu57_b3_47_l3457_45_l0n6_45_pl41n73x7}`

Since the key length is 10, and I luckily happen to know a plaintext prefix of length 10 of the flag, I can easily recover the key by XORing the encrypted text with the known plaintext. This leverages the fact that `plaintext XOR key = encrypted text`, so `plaintext XOR plaintext XOR key = plaintext XOR encrypted text = key`. After recovering the key, I can use the key to decrypt the rest of the message.

Since the flag might not start at the start, I rotate the encrypted_flag_hex to try every possible starting positions of the flag and search manually for the correct flag pattern.

(c)

Compile `code/6c.c` with `gcc -o 6c 6c.c -lssl -lcrypto` and execute it.
Flag: `NASA_HW11{https://youtu.be/1GxwDuV5JMc}`

This amazing code first builds a gigantic table of precomputed prefixes and their corresponding `i`, but the magnificent point of the code is that without sorting the table in advance and adapting binary search during lookup, the rate of PoW can only miserably reach around 40.

After building the table, the code initiates a connection with the server and quickly completes 10 PoW by looking up the table.

(d)

Flag: `NASA_HW11{y0u_KN0w_r3F13C710n_4774cK}`

Open up 2 terminals and both connect to the server. One terminal enters 5 while the other enters 6. The one entered 5 will get a nonce, then paste that nonce to the one entered 6. The one entered 6 will respond with a proof, then paste that proof to the one entered 5 with the name infront changed to someone else. And it's as easy as that.