CS161 Spring 2026 — Introduction to Computer Security — Discussion 11

Q1 *Suit of Armor Around the World* **(4 points)**

You are tasked with securing The Avengers' internal network against potentially malicious protocols! For each type of firewall and set of traffic, state whether the firewall is able to achieve the desired functionality with perfect accuracy. **Assume that IP packets are never fragmented.** All connections that are not mentioned can be either allowed or denied.

If you answer Possible, briefly (in 3 sentences or less) how the firewall should operate to achieve the desired effect. If you answer False, provide a brief justification for why it isn't possible.

Q1.1 (1 point) **Desired Functionality:** Block all inbound TCP connections. Allow all outbound TCP connections.

Firewall: Stateless packet filter

● Possible ○ Not Possible

Solution: This is possible by blocking all inbound packets with only the SYN flag set, which prevents inbound connections. This allows outbound connections by allowing outbound SYN packets, and the resulting inbound SYN-ACK packet is allowed.

Q1.2 (1 point) **Desired Functionality:** Allow all outbound TLS connections. Block all outbound TCP connections that aren't running TLS.

Firewall: Stateful packet filter

○ Possible ● Not Possible

Solution: While a stateful packet filter *can* reassemble a TCP data stream and look for signatures of a TLS handshake, it can still be circumvented with techniques such as sending multiple small TCP segments with the same sequence number but differing TTLs.

Q1.3 (1 point) **Desired Functionality:** Allow outbound DNS requests. Block inbound DNS responses. Assume that name servers always listen on port 53.

Firewall: Stateless packet filter

● Possible ○ Not Possible

Solution: This is possible (although it doesn't achieve much). One would allow outbound UDP datagram packets with the destination port 53 but block inbound UDP datagram packets with source port 53.

Q1.4 (1 point) **Desired Functionality:** Block all HTTP traffic that contains the literal string **Ultron**. Allow all other HTTP traffic.

Firewall: TCP proxy

● Possible ○ Not Possible

Solution: TCP proxies allow the TCP stream to be reconstructed exactly. Once the stream is reconstructed, the firewall can keep track of the entire HTTP request as state and, if it contains the string `Ultron`, drop the connection.

Q2 *Access Control* **(6 points)**

EvanBot uses `MailBot`, a web app thats helps summarize all the emails Evanbot gets in a day.

EvanBot needs to delegate access to `MailBot` through an OAuth token, only allowing `MailBot` to read emails and view EvanBot's GMail profile. Mailbot **cannot** send emails as EvanBot or delete emails

Mallory is an attacker who wants to exploit the MailBot's OAuth Flow.

Q2.1 (1 point) Select all that are true:

- ■ The OAuth Token MailBot recieves is an example of a capability
- ☐ If MailBot is compromised after receiving access, then the now malicious MailBot can delete EvanBot's emails
- ☐ This is an example of mandatory access control (MAC) because Google enforces the policy centrally
- ■ Evanbot can revoke MailBot's access without changing their Google password
- ■ The token follows the principle of least privilege

Solution: Knowing the token grants access to specific resources making it a capability. The token only grants access to read and view, so even when compromised, the scope of the token limits the malicious MailBot. This example resembles DAC more since EvanBot can specify the scopes ot grant each client like MailBot. The core usages of tokens is to grant access without leaking passwords, separating authorization from authentication. The scope per token also ensures least privilege.

The following steps occur when EvanBot wants to connect his Gmail to MailBot.

1. **Flow Initiation:** An user clicks "connect" on their client app, which redicts the user's browser to the corresponding authorization server.
2. **Authorization Request**: Authorization server recieves user's request along with the requested scopes and a redirect_uri.
3. **User Authentication**: The user logs in and approves access on the authorization server.
4. **Authorization Code Issued**: The authorization server redirects the user's browser to the redirect_uri with a short-lived one time code (auth code: `https://client.com/callback?code=AUTHCODE`)
5. **Code Exchange**: The client exchanges the code directly with the server for an access token.
6. **Account Linking**: The client stores the access token against the currently logged in user.

Q2.2 (1 point) At which step does a vulnerability exist where Mallory is able to inject herself into the process and cause a malicious authorization?

- ○ Step 1: Flow Initiation
- ○ Step 2: Authorization Request
- ○ Step 3: User Authentication
- ● Step 4:Authorization Code Issued
- ○ Step 5: Code Exchange
- ○ Step 6: Account Linking

Solution: Step 4: Authorization Code Issued

This is the only step where credentials are transmitted through the browser via a URL. Unlike the other steps which either happen server-side or require direct interaction with the authorization server, the auth code is exposed as a plain URI parameter that can be intercepted, saved, and replayed by an attacker.

Q2.3 (1 point) What exploit can Mallory do?

Assume:

- Evanbot is currently logged in to MailBot their browser
- Mallory also has a valid MailBot and GMail accoount
- Authorization codes issued are are valid until used
- EvanBot and Mallory are both active on StemEd where users can post and share links
- No State parameters are used in the Auth Flow

Solution: Mallory completes steps 1-3 on her browser with OAuth to get a redirect URL: `https://mailbot.com/callback?code=MALLORYSCODE`. Mallory saves this URL, intercepting the redirect so the code isn't used. Then she shares this link to EvanBot on StemEd. Since EvanBot is already logged in to MailBot, once EvanBot clicks on the link, EvanBot's browser will automatically attach his MailBot session cookie, triggering the token exchange. MailBot will get a token for Mallory's GMail, and store it linked to EvanBot's user account.

Q2.4 (1 point) What underlying attack is this exploit relying on?

- ● CSRF
- ○ XSS
- ○ SQL Injection

Solution: CSRF, because the exploit relies on the browser automatically attaching EvanBot's MailBot session cookie.

Q2.5 (1 point) Assume the attack from above succeeds, what is the end state?

- ○ Mallory is logged into MailBot as EvanBot and can read all his emails directly
- ● EvanBot's MailBot account is linked to Mallory's Gmail
- ○ Mallory obtained EvanBot's Gmail password through the access token
- ○ MailBot can send emails on EvanBot's behalf while controlled by Mallory

Solution: Since MailBot stores Mallory Gmail access token mapped to EvanBot's user account, when EvanBot uses MailBot, it will grab emails from Mallory to read then summarize to EvanBot.

The other options are false since Mallory never gets access to EvanBot's passwords (A/C are false). The token links to Mallory's gmail, so even if given access, any emails send will on behalf of Mallory.

Q2.6 (1 point) To defend again the attack above, an additional parameter is included in the authentication flow. When a client initiates an OAuth Flow, it will require a state parameter in the authorization request. The state parameter is a random, unguessable value sent along with the request and stored in the current user's session. The authorization server will echo this value back in the callback uri along with the AuthCode.

`https://mailbot.com/callback?code=AUTHCODE&state=RANDOMVALUE`

When the client recieves the callback uri, it will first check that the state in the URI matches the value stored in the current user's session. If they don't match, the request is automatically rejected.

Which of the following statements are true?

- ■ Even with state implemented, the attack would succeed if Mallory could read EvanBot's session data on MailBot's server.
- ☐ The state parameter is the main defense against the attack especially when EvanBot isn't logged in.
- ☐ The state parameter prevents the attack because it encrypts the authorization code in transit
- ■ If the state parameter is created from Hash(username), then attack earlier is still possible.

Solution: True: State relies on the being unguessable, if Mallory can simply read it, then Mallory can forge the URL to have the right state and the attack would work as before. False: When EvanBot isn't logged in, the attack fails regardless of state, since there is no session cookie, and no token exchange will be triggered automatically. EvanBot would be prompted to a log in page. False: The state doesn't encrypt, it is simply attached to the auth code for user verification. True: The secrecy of the hash input determines the strength of the state parameter. Since Hash(username) can be found and computed, Mallory can once again forge a valid URL.