Group Name: securecoders

Members:
Gavin Abramowitz
Luke Charlesworth
Michael Bartlett
Owen Wurst

P5 Writeup

Trust Model:

In this phase of the project, our group is going to widen the range of attacks that are possible against our system to expose the weaknesses in our cryptographic implementations that were covered by the trust models of previous phases.

- Authentication Server: The authentication server is no longer trustworthy. In this phase we can no longer expect data stored on the authserver to be secure from being leaked and viewed by potential attackers. For this reason, the authentication server may issue tokens to “imposter clients” if an attacker was to view the user names and passwords stored on this table and logon using another user’s credentials, allowing imposters full access into our system.

- Clients: We can no longer trust our end users to protect their passwords from attackers. End users are now expected to mistakenly reveal their passwords to attackers in the form of phishing scams and other deceptive schemes designed to trick users into giving them their passwords. As a result, we can again expect the authentication server to give tokens to “imposter clients”.

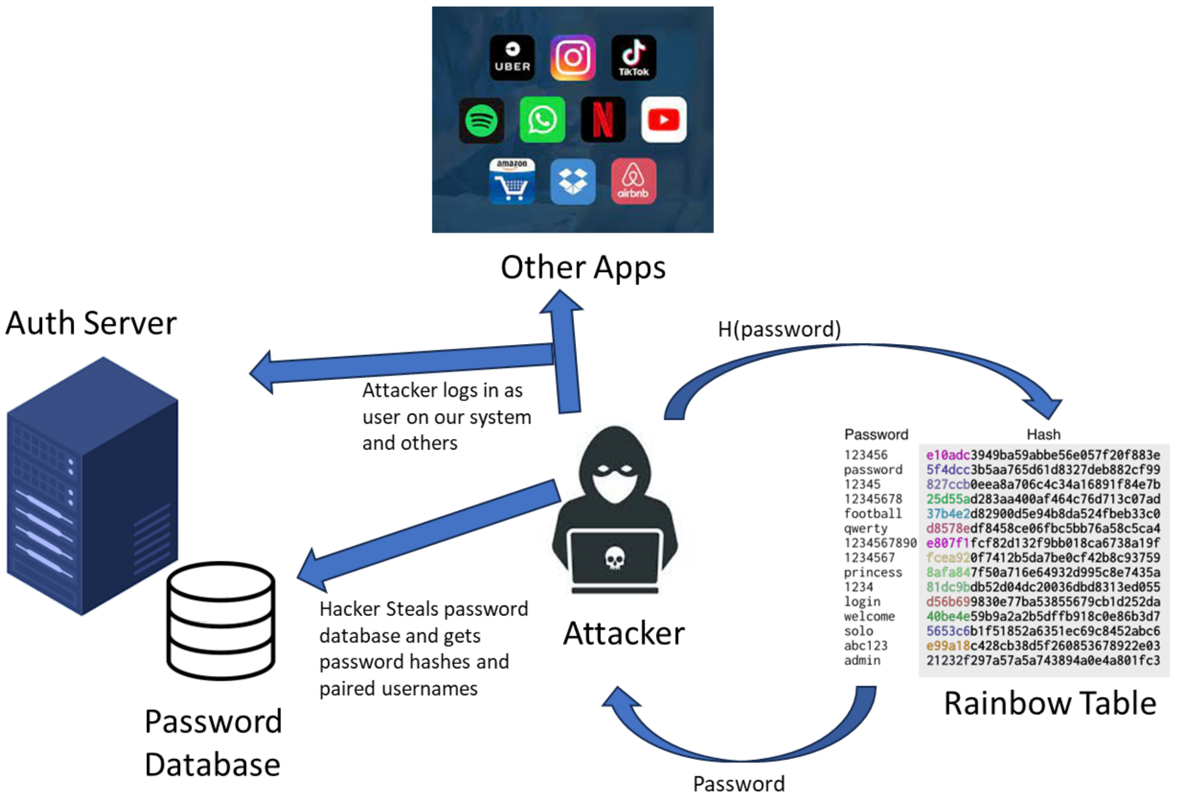
- Denial-of-Service (DoS) Attacks and Potential Service Disruption: All of our servers are now susceptible to denial of service attacks by attackers who may attempt to bring our servers down by abusing a resource disparity between the server and attacker. As a result, legitimate clients may not have access to our service.

Threat 1: Rainbow Table Attack

Threat Description:

In this updated threat model, we no longer can guarantee the safety of our auth server database, which contains our “users table”. This table contains the names, usernames, user ids, a log of their logins, and most importantly, the passwords of our users. In our current implementation, passwords are padded with 0s until they have 32 characters, and SHA-256 is used to hash them. When a user logs in, we hash the password they provided and make sure that it matches the hash that we have stored with their username. If the hashes match, the user is logged in and given an auth token. We thought that this was a secure means of storing passwords, but we now realize that this is vulnerable to a rainbow table attack.

A rainbow table is a giant table of password/hash pairs. These tables can be generated for specific hashing schemes, but because we are using SHA-256, an extremely common hashing algorithm, there are many tables already out there. Tables are pre-computed, so when a password hash is found, it can be looked up and the corresponding password found very quickly. Rainbow tables also take advantage of the fact that more than one password can hash to the same value, allowing them to provide an attacker with a password that works on the given system whether or not it is the user's original password. Our password database is a simple sqlite file, so if an attacker was able to obtain this, the opening of the file and viewing of the username-H(password) pairs is trivial. Once the password hashes are obtained, they can quickly be looked up in rainbow tables and it is likely that some of the hashes will yield results. For all of the users for which the attacker is able to obtain a password that has an equal hash to their stored password hash, there are three scenarios. In the best case scenario, users have used different passwords on all accounts, and obtaining this password will only allow the attacker to log onto our system with the user’s credentials. In a medium case scenario, the attacker is able to obtain a password with a working hash that is not the original password. In this case, the attacker is able to log onto our system and any system that has the same exact hashing scheme, provided the attacker is able to get the user’s username, which is generally trivial as they tend to be somewhat public, and the user has used the same password, which they tend to do. Finally, in the worst case, the attacker is able to get the original password from the hash and the user has used it elsewhere, in which case the attacker is able to log in as the user everywhere the user has used the password, provided they can get the username. This is a massive vulnerability because by using this hashing scheme for our passwords, we are endangering not only our system but we may be compromising our users’ bank accounts, medical records, and more.


Other Apps
Auth Server
H(password)
Attacker logs in as user on our system and others
Attacker
Hacker Steals password database and gets password hashes and paired usernames
Password Database
Rainbow Table
Password

Password	Hash
123456	e10adc3949ba59abbe56e057f20f883e
password	5f4dcc3b5aa765d61d8327deb882cf99
12345	827ccb0eea8a706c4c34a16891f84e7b
12345678	25d55ad283aa400af464c76d713c07ad
football	37b4e2d82900d5e94b8da524fbeb33c0
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567890	e807f1fcf82d132f9bb018ca6738a19f
1234567	fcea920f7412b5da7be0cf42b8c93759
princess	8afa847f50a716e64932d995c8e7435a
1234	81dc9bdb52d04dc20036dbd8313ed055
login	d56b699830e77ba53855679cb1d252da
welcome	40be4e59b9a2a2b5dffb918c0e86b3d7
solo	5653c6b1f51852a6351ec69c8452abc6
abc123	e99a18c428cb38d5f260853678922e03
admin	21232f297a57a5a743894a0e4a801fc3

Solution and Defense of Efficacy:

The solution to this attack has a couple parts. A specific rainbow table contains password-Hash(password) pairs for a very specific hashing scheme. Since we use SHA-256, a common scheme, there are lots of tables out there with the ability to "decrypt" our passwords. The first thing to do is to add a specific salt, which is a random value to all of our passwords prior to hashing. Since rainbow tables rely on hashing "common" passwords, a rainbow table for our specific salt would have to be found, or more likely, generated, to complete a rainbow table attack; however, if we take this strategy one step further and generate a new random value for salting every password, no rainbow table could be generated that could compromise all of our users because even if two users have the exact same password it would not have the same hash. Now, any rainbow table attack against our system would require obtaining the password database and essentially generating an entirely new rainbow table for each unique salt. While this is not feasible for most people, it is possible that if our system is storing extremely valuable sensitive documents or one of our users is extremely high-profile, that there might actually be the incentive to take the password database and generate a rainbow table specific to the salt of the user. To add some protection against this, we will no longer be using SHA-256 to hash passwords. SHA-256 is a very secure algorithm but its speed does allow the possibility of offline attacks for something as valuable as passwords. Instead, we will use Argon2 for our password hashing which is specifically designed to be slower and more memory intensive. Argon2 also is configurable, and can be tuned to further decrease performance and limit utilization of parallelism. By using Argon2, we make the possibility of generating a new rainbow table infeasible by making it extremely computationally intensive. With the addition of unique salts for every user and the switch to Argon2, we guarantee strong security against rainbow attacks.

Threat 2: Phishing

Description Of Threat

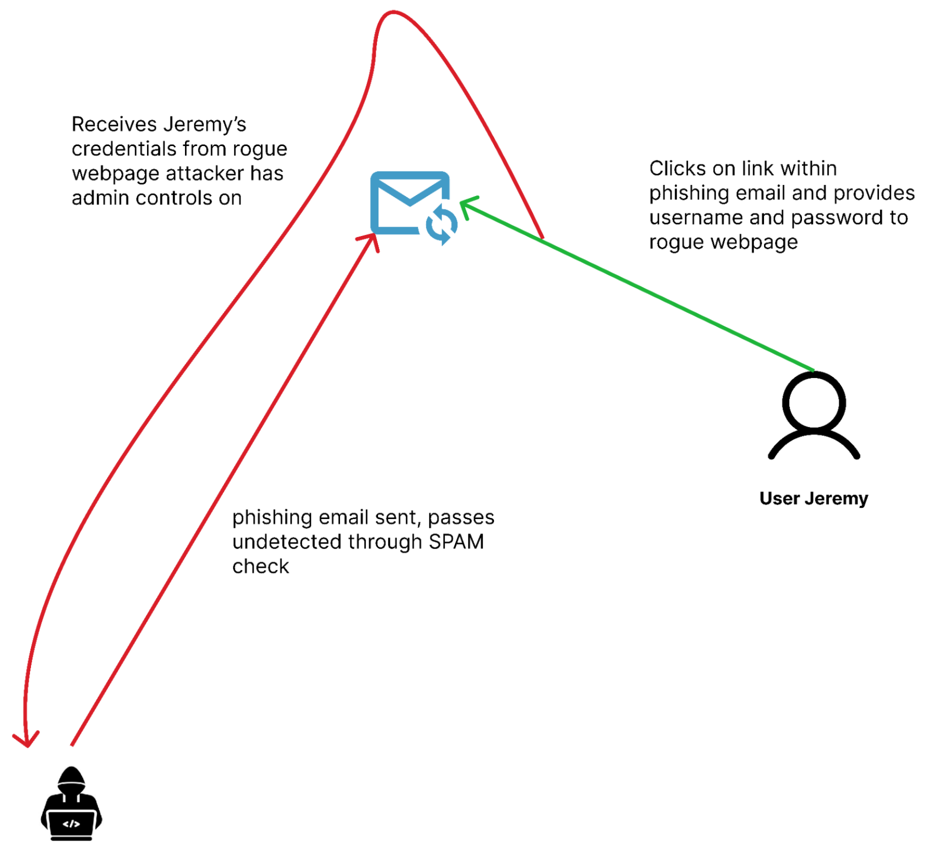
Phishing is a type of cyber threat in which attackers utilize deceptive psychological tactics to trick individuals into providing sensitive information, typically related to login credentials, credit card numbers, or other forms of personal data. “Phishing” got its name as an analogy to fishing, because attackers will cast their bait (i.e. a deceptive email, text message, or phone call) to a wide variety of targets, just like how when you fish you hope to throw your bait into a body of water with tons of potential fish to catch. The goal here is to lure as many unsuspecting victim user’s as possible into providing valuable information. Typically, attackers will attempt to make their phishing message’s iconography and branding similar to that of the service they want to gain unauthorized access to; thereby tricking the victim user into believing this is a legitimate and sometimes urgent message from that service, so they provide their valuable information without suspecting a thing. Phishing attacks have only grown more popular in recent years, because they are essentially immune to online service’s internal security measures, as they circumvent attacking the system itself by instead attacking the ignorance and inattentiveness of that system’s users, and then use what they have found to impersonate that user effectively, stealing whatever else they can.

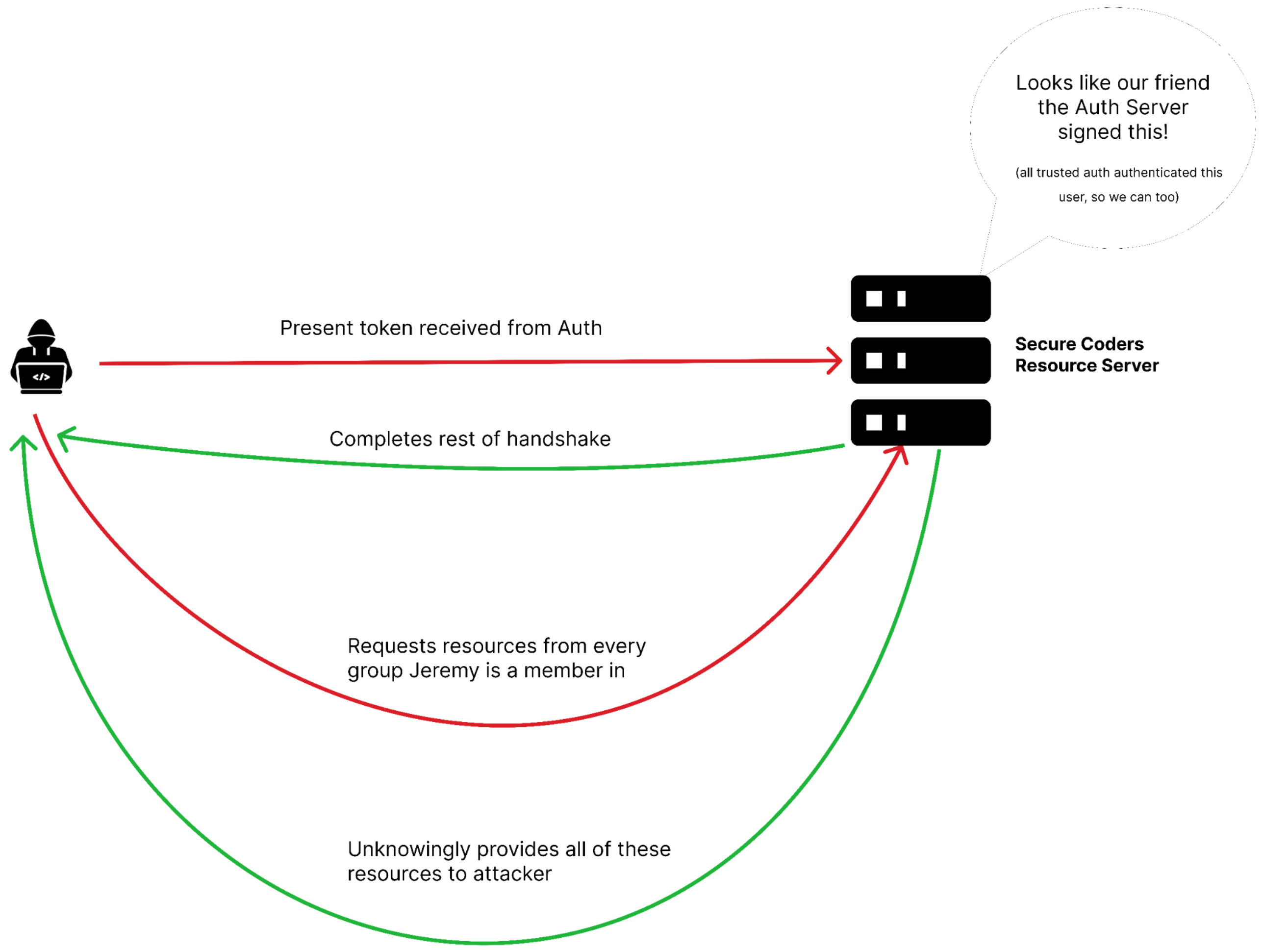
Illustrative Attack Example:

Alice is a member at Chase Bank, where she stores the majority of her liquid capital. She is scrolling through her inbox one day when a new email appears with the title: “Urgent Notice from Chase Bank, 10 hours left to save your account!” Shocked by the message, Alice quickly clicks on the email and reads a notice from what she believes to be her bank stating that their new policy mandates 2-Factor Mobile Authentication, and at the end of this 10 hour time period they will be closing all accounts that have not set up this new authentication method. Thankfully, the email contains a link: “Click here to set-up 2-Factor Mobile Authentication”. Alice lets out a sigh of relief as she thinks she can fix this right now and save her account; so she follows the link to a webpage that looks identical to that of Chase Bank’s Authentication Screen. Typing fast and suspecting nothing, she enters her Account #, Routing #, Username, Password, and Phone #, and then hastily performs a text-message confirmation with the webpage. A success screen pops-up saying: “Congrats, 2FA is set up and your account is saved!”. Alice is happy and goes back to scrolling in her email.

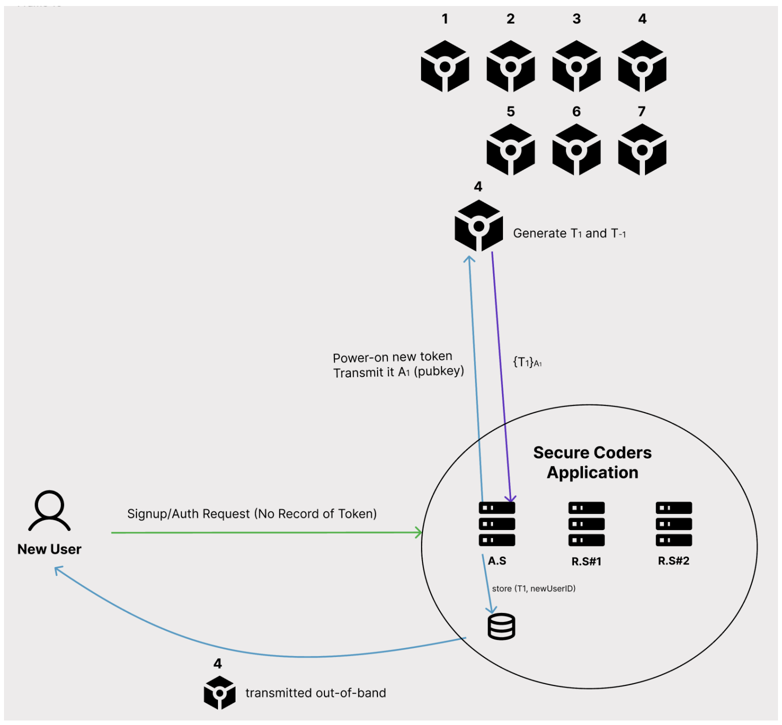
Meanwhile, the attacker group: “PhishForFools” has landed yet another victim, the fifteenth of this week! Alice has given them literally all of her bank credentials thanks to their perfectly executed phishing email and mock-site. They simply download the Chase Mobile App, login as Alice using the information she provided, and then complete a wire transfer of her entire account balance to a rogue account #. This bank account is merely a shell for untraceably funneling funds into an offshore secretive account where the money eventually ends up and is divided among the "PhishForFools" group. They celebrate and toast to Alice's obliviousness. Alice soon learns of this depressing news after a grocery store shopping trip goes horribly wrong. She is now a teary wreck, projecting her anger at the Chase Customer Service representative who keeps desperately reiterating to her, "I am so sorry Alice, but it looks like you transferred the funds yourself this morning".

<u>How Threat could Manifest in Your Current System:</u>


Receives Jeremy's credentials from rogue webpage attacker has admin controls on
Clicks on link within phishing email and provides username and password to rogue webpage
User Jeremy
phishing email sent, passes undetected through SPAM check


Looks like our friend the Auth Server signed this!
(all trusted auth authenticated this user, so we can too)
Present token received from Auth
Secure Coders Resource Server
Completes rest of handshake
Requests resources from every group Jeremy is a member in
Unknowingly provides all of these resources to attacker

<u>Countermeasure Mechanism:</u>
Our countermeasure mechanism to mitigate this threat relies on the use of physical authentication tokens, public cryptography, and time-sensitive codes:


1
2
3
4
5
6
7
4
Generate T_1 and T_{-1}
Power-on new token
Transmit it A_1 (pubkey)
$\{T_1\}_{A_1}$
Secure Coders Application
New User
Signup/Auth Request (No Record of Token)
A.S
R.S#1
R.S#2
store (T1, newUserID)
4
transmitted out-of-band

Onboarding

1. On first signup, new users will be presented with a physical authentication token and instructed to keep this token safe, hidden, and always on their person when they intend to use our application. These tokens will be battery powered, rechargeable, always on, and will

have only one button that says: “Press for Code”.

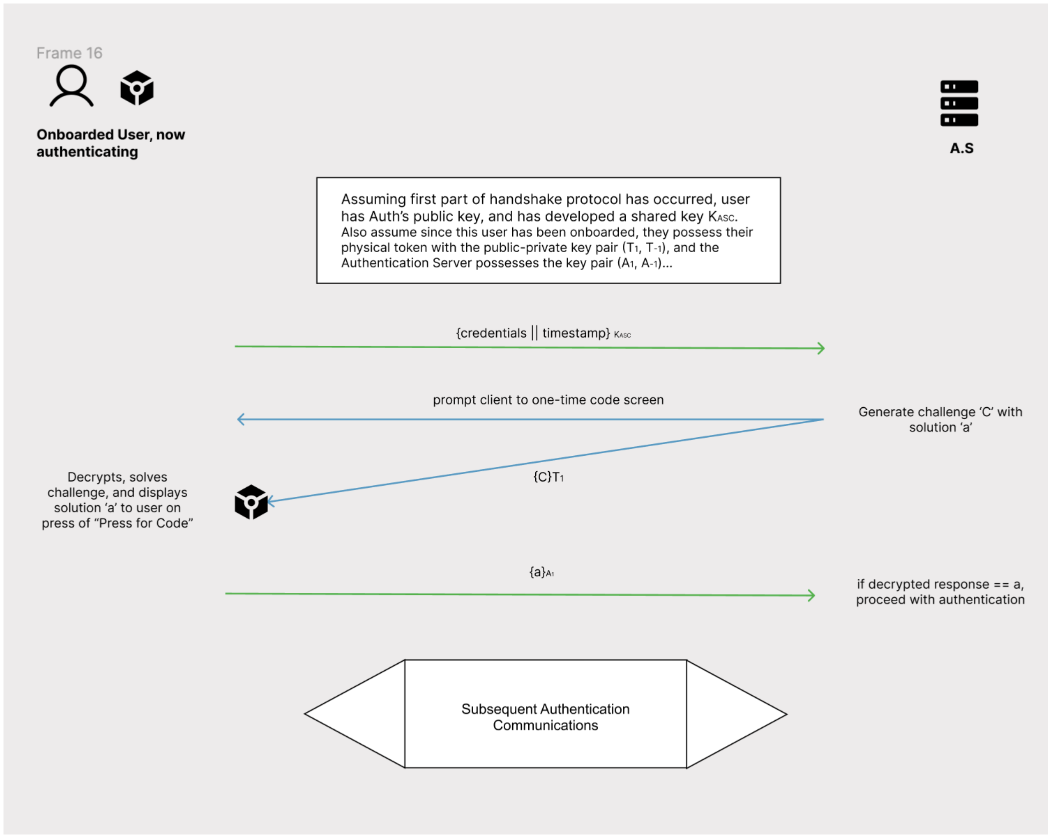
2. Each token will be a self-contained machine with the ability to communicate across a network to our Authentication Server. On creation, the authentication server will first provide the token with its public key. Then the token will generate a 4096-bit Public and Private RSA Key Pair and then securely provide the Authentication Server with its Public Key (encrypted with the Authentication Server’s Public Key so only this server itself can decrypt it). The Authentication Server will then receive its message and add a column entry in its Users’ table for each user’s token’s public key. Note that these keys will never be regenerated so the token will never change its public-private key pairing; hence the reason for
the large key size as a security enhancement measure.

Authentication Requests:

1. If any new user attempts to sign-in and the Authentication Server has no record of their physical token, it will direct them through this aforementioned token onboarding process before they can even try to authenticate.
2. For onboarded users with physical tokens, there will now be an additional step to our authentication handshake protocol. After client’s transmit their credentials, timestamp, and proposed shared session key to the Authentication Server, the user’s screen will shift to a page with a text input for their token’s one-time code. Along with two more buttons, one for “Confirm”, and one for “Send New Code”.
3. As this new interface is being displayed, the Authentication Server will be sending some unique challenge ‘C’ (encrypted with the user’s token’s public key) directly to the physical token itself. The token will then decrypt this message and solve the challenge efficiently.
4. By the time the user clicks the “Press For Code” button on the token it can be assumed the physical token has solved the challenge. Thus, the one-time code it generates will be based on the solution of the challenge it just received, and will always be between 15-20 ASCII characters. The reason for this choice was to prevent a brute-force attempt attack on the one-time code (additional brute-forcing safeguards will be mentioned later). Also note that if no challenge has been received by the token or even if some message was received but it could not be decrypted with the token’s private key the token will not generate any response.
5. From the moment this challenge response appears on the user’s token, this user will have 45 seconds to type this challenge response into the webpage and press “Confirm”. If they miss the timeframe they can click “Send New Code” And steps 3 & 4 will be repeated with another uniquely generated challenge from the Authentication Server.
6. Once the user has typed in the challenge response and confirmed, the client will then encrypt this response with the Authentication Server’s Public Key and transmit it to the Authentication Server. The Authentication Server can then verify that this challenge response is correct, indicating that the user is in possession of the physical token they

have on record, in which case they will proceed with the rest of the authentication handshake protocol.

7. In the event an incorrect code is transmitted more than once by the client to the server, the Authentication Server will assume this client is malicious and block any future requests from this user's account for 48 hours while SecureCoders Tech team investigates the situation further.


Frame 16
Onboarded User, now authenticating
A.S
$\{\text{credentials} \,||\, \text{timestamp}\}_{K_{ASC}}$
prompt client to one-time code screen
Generate challenge 'C' with solution 'a'
$\{C\}T_1$
$\{a\}_{A_1}$
if decrypted response == a, proceed with authentication
Subsequent Authentication Communications

Assuming first part of handshake protocol has occurred, user has Auth's public key, and has developed a shared key K_{ASC}. Also assume since this user has been onboarded, they possess their physical token with the public-private key pair (T_1, T_{-1}), and the Authentication Server possesses the key pair (A_1, A_{-1})...

Decrypts, solves challenge, and displays solution 'a' to user on press of "Press for Code"

Security Proof:

To prove that this proposed mechanism sufficiently addresses this threat of phishing, let us consider what would happen in the event of a phishing attack. Even if a given user is successfully "phished" into providing an attacker with their username and password credentials, the attacker would not gain possession of this user's physical authentication token. This token is on the user's person and thus cannot be transmitted across an online channel; and users were specifically instructed to keep the token safe and concealed. Thus, when this attacker attempts to login with the credentials they have successfully obtained, they will inevitably reach this new authentication step where they must enter a valid [challenge solution] one-time code into their client application page. Since they do not have any knowledge of the challenge or possession of the token to receive the code, all they can perform is a brute-force attack on the one-time code's potential values. However, each challenge solution code only lasts for approximately 45 seconds and after 2 unsuccessful attempts, the user's account is temporarily blocked and flagged; thus, there is no efficient way to brute-force this one-time code, and the attacker will be detected by our application. In summary, the attacker will never possess the user's physical token or its private key (because this key was also securely generated and never sent unencrypted), which means they will never have the knowledge necessary to provide the one-time code to the Authentication Server and will not be able to move past this stage in authentication to conduct any malicious activities. Also note that even if the attacker somehow did obtain a prior one-time code after a user's session, these codes are temporary and only valid for 45 seconds, so the Authentication Server would not validate this old cold on a new login attempt.

Correctness Proof:
The correctness of this mechanism rests heavily on the initial user onboarding step, in which the new user is given their physical token. As long as their token is correctly set up such that the Authentication Server has a record of its public key, and the user uses the token as our Application describes; they should have no problem clicking the "Press for Code" button and typing in the code their token gives them. If the user loses their physical token this mechanism will fail, but it is our sincere hope that we educate our users on the importance of their token to mitigate this risk.

Threat 3: Denial of Service

Threat Description

A Denial of Service (DoS) attack can occur when a malicious client has the ability to overwhelm a server causing service to be prevented to legitimate clients. These attacks typically occur when a malicious client abuses a resource disparity. Typically, when there is a resource disparity, it is not because the server has limited resources to perform its typical tasks, rather there is an issue in the design of the system that allows for a small expenditure of resources to cause a large expenditure of resources. In other words, a malicious client causes typical requests that the server would handle to cause an amplification of resources being used.

For example, a DOS attack could happen when a client generates many network requests in a short amount of time, causing the server to attempt to handle more requests than it has the resources for. Another example where a client generates some random application data that the server has to decrypt. This would cause the server to perform many decryption cycles, leading to the server wasting tons of resources on decrypting nonsense data and being overwhelmed in the process.

DoS attacks are seen frequently in the news and are known to take down many large companies. In recent years DoS attacks are carried out as a Distributed Denial of Service (DDoS) attack. The key difference between DoS and DDoS attacks is that, in a DDoS attack, many malicious clients attack a system at the same time as opposed to a single malicious client attacking a system. These attacks have only been growing and in October, Cloudflare, a company that offers services for cybersecurity, content delivery, among other solutions, even said that in the third quarter of 2023 they saw the most DDoS attacks ever recorded in a quarter. When developing a server, it is important to have mitigation strategies in place to prevent these attacks.

How the Threat Could Manifest in our Current Implementation:

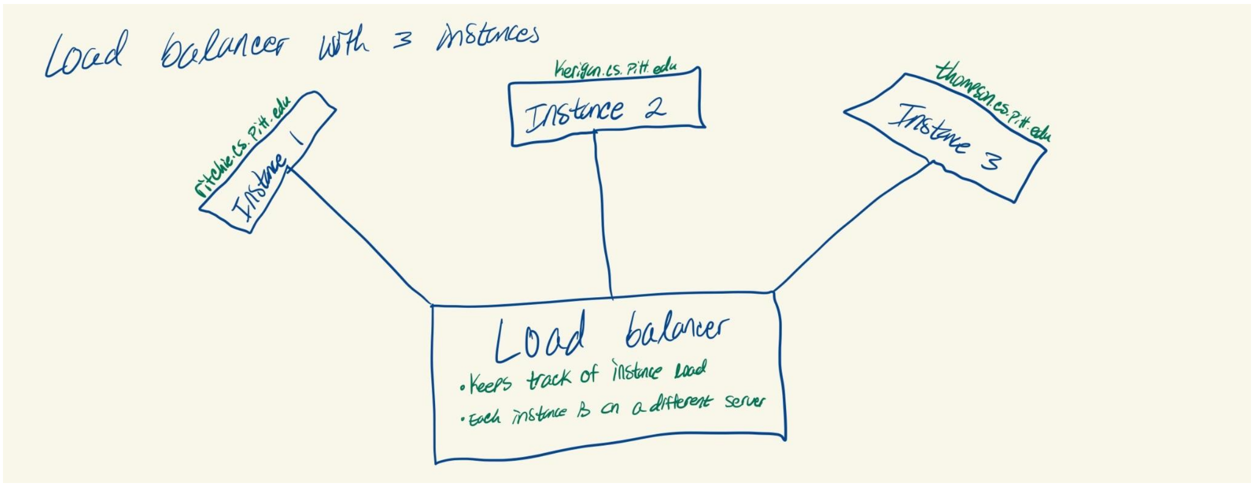
One way a Denial of Service attack could occur in our current system is by an adversary carrying out a volumetric attack on our system. This could be carried out by the adversary continuously sending garbage to our API routes on both the Authentication and Resource Servers. Currently, routes on the server are not rate limited. As a result, an adversary could flood the server with requests in order to make it do more work than intended, causing it to crash and run out of resources under the weight of this attack.

A DoS attack could also be carried out by an adversary forcing either server to perform many cryptographic operations, leading to a resource exhaustion attack. This attack could occur if an adversary authenticated themselves and rapidly made requests to either server. Because all requests must be decrypted by the server and responses must be encrypted, a DoS attack could be made by a client that appears to be totally legitimate.

Countermeasure Mechanism:

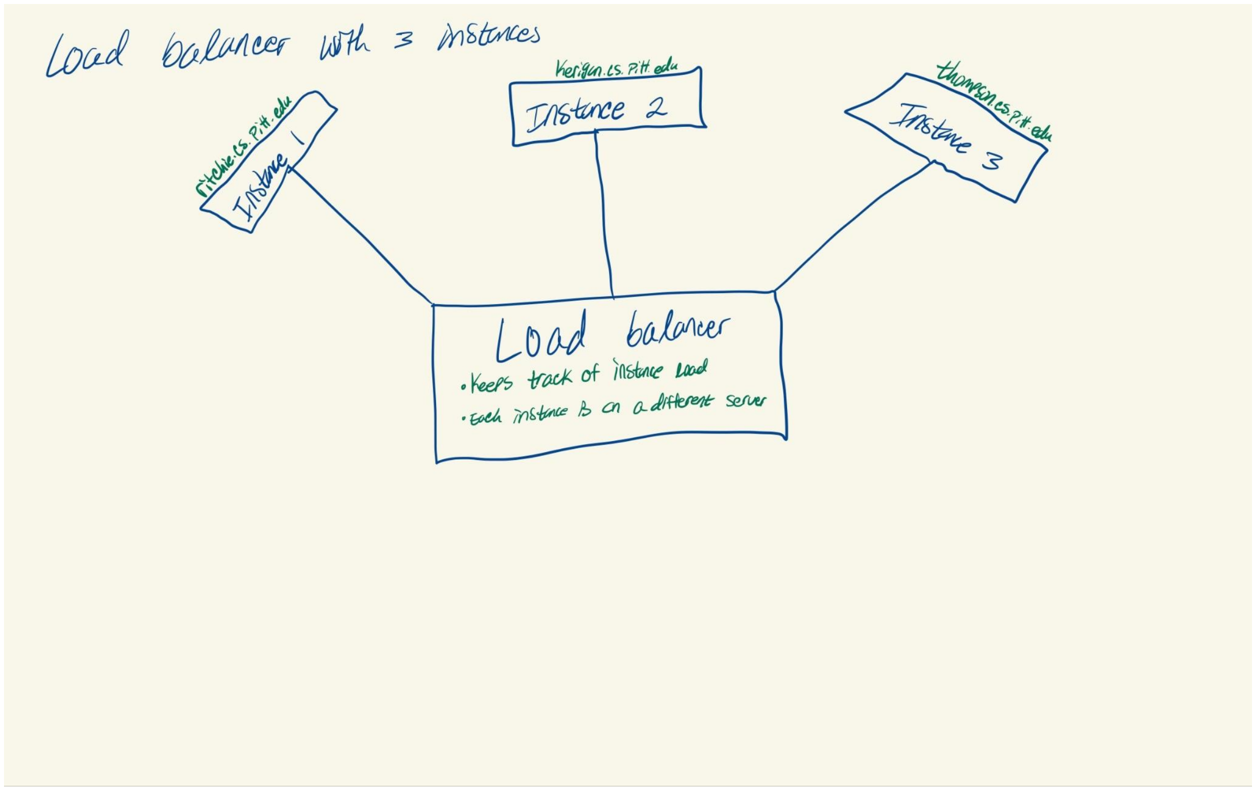
In order to mitigate the risk of DoS attacks, rate limiting, IP address blocking, load balancing, caching, and cryptographic puzzles will be implemented. Rate limiting will introduce a limit on all API routes. For routes that are less resource intensive and/or more commonly requested, a rate limit of 20 requests per second per IP address will be implemented. For routes that are more resource intensive and/or less commonly requested, a rate limit of 50 requests per second per IP address will be implemented. This should allow our server to disallow too many requests at one time and properly balance resources in combination with the other tools being implemented. If one IP address is continuously being rate limited it will be blocked from making requests to our server.

Load balancing will be implemented in order to allow multiple instances of each server to run concurrently. The load balancer can assess how many requests each instance is handling and send each request to the server with the most resources at that moment. If an instance is being overloaded with requests and is taken offline, other instances can still handle requests. Additionally, if an instance was taken offline as a result of one user’s actions, their IP address will be blocked from making requests to the system.



Caching will be implemented in our server instances. Caching will allow requests that return common responses to be stored. For example, if multiple clients request a list of files for the same team in a short period of time the request will be cached and can quickly return the information to the user. This will allow less resources to be used as the information can be quickly retrieved from the cache and a database call will not be needed.

Finally, cryptographic puzzles, specifically on unauthenticated routes, will be implemented. These puzzles will be easily computed by the server and slow the client down from making requests. If some user tries to request an unauthenticated route they will be forced to solve a cryptographic puzzle before receiving the response. For example, if a user requests the resource server's public key (a route that only requires a user ID in the body), the resource server will send back a hash inversion puzzle. The client will then solve the puzzle and send back their answer. If the answer is correct then the user will receive the public key for the resource server.



<u>Proof of Efficacy:</u>

Rate limiting and IP address blocking will allow for users trying to exhaust resources to be blocked from using our system. Once a user is rate limited and their IP address is blocked, they will be blocked from using the system. If a user tries to subvert the rate limiter by logging in on clients on other IP addresses, their user ID will be blocked from the system. As a result, malicious users will be limited in their attempts to make malicious requests.

Should an instance of a resource server be overloaded, load balancing will still make the server available to other users trying to make requests. The server will stay available to other users because it can send requests to the non-overloaded instances. These instances will be placed on different machines so that each instance is not dependent on another instance's resources. While this does not totally eliminate DoS attacks, this helps to alleviate the stress of them should they occur.

Caching will help to improve response time and alleviate stress on the server's resources. If common responses are cached by the server, this could provide a much faster lookup time for information that needs to be returned. If the information is stored in a cache and can be returned quicker, less resources will be used resulting in less stress on the server.

Cryptographic puzzles will slow the client down when making unauthenticated requests. Because authentication tokens prove that a user is who they say they are, authenticated requests are protected, leaving unauthenticated requests to be more vulnerable to an attack. When a client has to solve a cryptographic puzzle, their request time from the client to the server will become slower. As a result, the client will make less requests to the server overtime, causing the server to spend less resources on a possible attack.