|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | LOW | MEDIUM | MEDIUM-HIGH | HIGH | CRITICAL |
| Very Likely |  | Phishing [CSI.3] |  |  | Database Leak [CSI.1] |
| Likely |  |  |  | Malware [CSI.6] |  |
| Unlikely |  |  |  |  |  |
| Very Unlikely | Brute Force  [CSI.5] |  |  | Cryptographic flaw [CSI.4] | Denial of service [CSI.2] |

## References to the risk analysis table

### CSI.1 The threat of a database leak is always critical as it can be a danger to life as the information dealt with is confidential, to minimize the risk of this point of failure implementing the database system locally rather than dealt by a third-party cloud provider is advised

### CSI.2 Having a denial of service can disrupt this service as it is an online interaction implementation, a necessary process to implement is to have the critical operations a.k.a backend firewalled off where all interactions that must be exposed is to the minimum. Modular approach would have the client endpoint distributed via CloudFlare or other C.D.N1 which will reduce the likelihood of this incident occurring; implementing using Google’s reCAPTCHA will be effective in rejecting access from bot requests but allowing human requests which reduces most spam

### CSI.3 This can happen, the client we do not expect to have a very strong understanding of security principles but at least a standard. The site of entry will be encrypted in which the HTTPS protocol is used rather than HTTP as the site will required an TCP/SSL certificate of validation. This will be something which ensures that the client connecting is connecting the valid service

### CSI.4 Having poor encryption technique can ruin the whole operations to a halt, malicious actors using malicious code can hijack the backend system. Having every component sandboxed with a unique encryption handshake requiring a key is necessary.

### CSI.5 Incident can be prevented by rate limiting, having a limit for the users on login attempts. This means to log the attempts and have grace period for each login token attempt using a token.

### CSI.6 This incident is very possible, as the infrasturcutre requires an internet connection throughout this cannot be treated as something of a low threat, the mitigation necessary will be security policies enforced which most operation is under a strict policy so that system resources require an authentication to go down before being altered by any means

I will be producing a STRIDE threat model table for classification of the threats of the proposed system of the equipment rental pool distribution system

Below I have keyed for which I will format this threat model

Key:

STRIDE – S [Spoofing]; T [Tampering]; R [Repudiation]; I [Information Disclosure]; D[Denal of Service]; E[Elevation of privilege]

S – being able to fool someone or something

T – Maliciously messing with something

R – Denying something, this can only be exploited when evidence is not available

I - Having confidential information shared to people who are not allowed to know

D – Malicious actions to disrupt the service of a system to serve the legitimate purpose rendering it useless for intent

E – Having access to perform actions which are not supposed to be viable without a clearance level normally

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Threat | Threat Description | Element Of Failure | Interaction Level | S | T | R | I | D | E | Likelyhood |
| CR1.1 | Credit Hijacking | Database | Software (SQL Services) |  | X | X | X |  | X | Very Low |
| FE1.1 | Denial of service | Back-end Server | Software (Firewall) |  | X |  | X | X | X | Moderate-High |
| FE1.2 | Misconfigured Front End Leaks | Cache Server (front-end) / CDN Failure | Software (Console)  Hardware (Storage) |  |  |  | X |  | X | Low |
| CR1.2 | Cloned Login | Back-end Server | Software (Authentication) | X | X | X | X |  | X | Moderate |

With regards to these threats identified, I will clarify the damage they can cause and mitigations for such flaws.

CR1.1 – Due to this system in place being a reward based, records of each transaction is stored on a database, when the database interactions are live operation between the client and the database malicious code can be used this is known as an SQL injection attack, to counter this issue the most suitable approach would be to have the database access on read only to the clients with the administration privilege handled by the clerk as well as to not allow queries to be executed manually by the operations of the customer lender and the customer borrower.

FE1.1 – This attack would mean the service comes to a complete halt, it would mean that no business operations could occur whilst service is down. This would be due to the back-end server which shouldn’t be accessible to the public being under heavy disruptive load preventing all legitimate requests happening. The first counter for this issue would be to make the client (customer lender & customer borrower) have no interaction necessary directly to the back-end meaning all interaction is done through a front end which deals with the back end. This would prevent most malicious actors to not even be able to probe the back-end to initialize a DDOS (distributed denial of service) attack to the service however this solution is not enough as confidential information can be leaked by the maintenance (clerk) to a bad actor. This means firewalling and configuring the back-end to only deal requests via a whitelist rather than blacklist approach is necessary. This means only confirmed legitimate requests will use system resources from the back end and everything else will be dropped.

FE1.2 – From following onto a prevention to FE1.1 [Denial of Service]; adding more elements of infrastructure to where confidential information is stored means more points of failure, having a front-end cache server to stop the client (customer lender and customer borrower) from having access to probe the back-end can cause another issue, this issue is when logged in (cookie data) of confidential information is cached and relayed into the service for non-authenticated individuals. This reliance on third party infrastructure can create an additional layer of data integrity breech.

One famously documented incident of FE1.2 from misconfigured front end was with the CDN CloudFlare in which information which wasn’t supposed to be cached was cached, “On Thursday night, Cloudflare publicized a major bug that existed for several months in content delivery network — how web pages are loaded for users based on their location — that allowed private information like passwords, emails, and private messages to be scraped and cached by search engines like Google.” (Peter Hess, Inverse.com, 2017); as this flaw is with the element of service used to mitigate the previously mentioned FE1.1 flaw it is inevitable it could arise but this does not mean we cannot do enough to atleast minimize this attack surface of potentially exposed data. This can be reduced by reducing the amount of time content is cached meaning cache flush occurs multiple times which does mean more traffic requests for back-end although still with the goal to not allow illegitimate traffic through.

CR1.2 – similarly due to the incident at FE1.2 when a user is authenticated a login token is generated so that the user isn’t automatically logged out (having to sign in for each page load) this means injecting a cookie onto the browser/app; incidentally this means that this “cookie session” token can be used as proof a user is logged in but this can be used maliciously if a users token is phished by a bad actor, the bad actor could essentially clone the cookie by injecting it to their own system and can impersonate the victim, this could be done without proof of the event even happening. The bad actor could deny proof of this because cloning a cookie does not generate a new login token identifier meaning no record will be amended to user authentication logs.

Firstly, to combat this flaw one approach would be to abolish a cookie system, however this is not convenient at all so this is a no-go. The most acceptable approach would be to limit the lifespan of the cookie with a set of rules to invalidate token after a set time or whenever the session is closed between the client and the service via their browser.