Tampere University, Unit of Computing Sciences
COMP.SE.610 Software Engineering Project 1
COMP.SE.620 Software Engineering Project 2

Group 1

Get a Room!

Test Report

Jere Koski
Joona Prehti
Joonas Hiltunen
Martti Grönholm
Mikko Pirhonen
Sara Brentini
Vivian Lunnikivi

Version history

Version	Date	Author	Description
0.1	6.12.2021	Vivian Lunnikivi	Copy template, fix layout, and add team details.
0.2	15.12.2021	Vivian Lunnikivi	Write sections 1.1 and 1.2.
0.3	15.12.2021	Mikko Pirhonen	Ch. 2 beginning, System testing plan
0.4	15.12.2021	Vivian Lunnikivi	Add sections 2.1, 2.2, and 2.3.
0.6	16.12.2021	Sara Brentini	2.4, 3, 4.7 and added appendixes on usability testing
0.7	16.12.2021	Vivian Lunnikivi	Add sections 1.3, 2.1, 2.2, 2.3 and 2.5.
0.8	17.12.2021	Joonas Hiltunen	Write section 4.4.
0.9	17.12.2021	Sara Brentini	Proof read and fixed typos.
1.0	17.12.2021	Joonas Hiltunen	Add appendix C, added text to section 4.4 and fixed typo in references.

Table of contents

1 Introduction

1.1 Purpose and scope of document

This document summarizes the testing approaches and test results for *Get a Room!* software system, implemented as a student project and offered for implementation on the Tampere University courses *COMP.SE.610* and *COMP.SE.620 Software Engineering project 1 & 2* by the technology company Vincit in the fall 2021.

Testing the system included requirement validation, unit tests, regression, acceptance, ad hoc usability, and exploratory testing (no formal cases but feature release notes as a reference) by the client, as well as manual system testing by the developers, supplemented by a formal usability test session as well as security review. The details and results of the varying testing approaches are explained in this document.

In general, the delivery team is happy with the product quality in terms of usability, stability, security, and extensibility. The extensive efforts put into testing the system throughout the project has surfaced a multitude of issues from requirements to implementation, many of which the team was able to fix during the fall. Testing and validating the system and its requirements has also provided insight and support on further development of the system, which the team hopes to communicate for possible future developers in form of documentation and testing pipeline.

1.2 Product and environment

The product under testing is a mobile-first progressive web application (PWA) called *Get a Room!,* constituting of a React frontend application with a Node.js backend, connected to Google Workspace cloud API. The purpose of *Get a Room!* app is to make reserving meeting rooms on the fly easy, and its targeted end users are the employees of Vincit.

Supported platforms include both mobile and desktop environments, and official support is provided for smartphones running recent versions of iOS with Safari, or Android with Chrome. Users log in the system using their Google IDs and make reservations to meeting rooms located at their preferred office location.

1.3 Project constraints related to testing

The most critical constraint concerning testing was the availability of resources. To ensure the product matched user needs and worked properly required a diverse set of testing techniques and simply, a lot of time. Therefore, towards the end of the project, more and more emphasis was put towards testing and fixing issues revealed by testing.

From the technical point of view, testing environments, user data confidentiality and data integrity posed some restrictions to testing. For example, the customer had to perform acceptance testing and ad hoc usability testing in the testing environment with test credentials instead of the production environment, where end users could use their personal google IDs for sign in. This harmed evaluating the onboarding experience, since users had difficulties with signing in with the test credentials, when they had their own Google IDs saved in their device's memory.

Usability testing had also taken place at Vincit's premises, for making the testing situation as similar to real use situations, as possible. The customer had also requested for a

week's notice to ensure enough end users can attend the session and that the customer-provided testing gear was available for use during the test session.

Constraints related to the usability testing session also included the confidentiality of test users' personal data. The confidentiality was temporarily endangered since we had to film the user's device screen during the session to evaluate user experience. The participants were be notified about filming and asked to turn notifications off for the duration of the test session.

1.4 Definitions, abbreviations, and acronyms

Exploratory testing	no test cases but story to follow
Test	testing of one requirement, functionality, or feature
Test case	everything required to one test
Test suite	a set of tests and test cases, e.g., for usability testing.
Feature branch	A feature branch is a source code branching pattern where a developer opens a branch when she starts working on a new feature [1].
Code review	Code Review, also known as Peer Code Review, is the act of consciously and systematically convening with one's fellow programmers to check each other's code for mistakes. [2]

2 Testing process

Code style and general quality in the codebase are achieved with a combination of multiple methods and tools. Automated tools are preferred over style guides and conventions to make development easier. Manual code reviews are also utilised, since they cover a different set of errors compared to other static analysis tools.

Uniform code style throughout the project is achieved using Prettier [3], which is a code formatting tool that formats the source code in the project using a pre-commit hook. In practice this means that whenever new code is pushed into the remote GitHub repository, it is already formatted.

A linting tool called ESLint [4] is utilized to catch programming errors and defects in the developed code. It is run as a part of the CI/CD pipeline to ensure that new code in the repository is of high quality and contains no obvious errors.

Finally, code reviews have been performed as a final part of the process of developing a new feature. Code reviews were done after a feature was developed in a feature-branch, with the CI/CD pipeline passing without errors (Linting and test cases succeeded). After a passing code review, the feature could be merged. Code reviews have proven to be an effective way of catching programming errors which static analysis tools have a hard time catching, such as logic errors or inefficiency.

2.1 General approach

The testing approaches on the high level was to test throughout the project, focusing always on the work items on the team's table. Both customer, developers and end users participated in the testing activities.

First, testing targeted the requirements in the form of discussing and approving the collected requirements and their priority with the client. This work was orchestrated by our UX head. Once implementation work started, unit tests were written by developers to ensure correct functionality and prevent regression, roughly at the same time as the features were delivered. The unit tests are also integrated in the CI/CD pipeline, configured by our CI/CD specialist.

Related to feature releases, the customer got to perform acceptance testing towards the end of the project. The team agreed to post release notes to Slack every time new features were merged into the staging environment, where the customer could try out the feature and give feedback. On the customer's acceptance of a feature, features were then released to the production version of the product.

Since the timetable for the project was rather tight, we could not implement integration or system tests in the form of automated tests, as we had hoped in the beginning. However, all features were tested manually, first by the developers, then by the client on our staging environment before acceptance. In addition, informal manual system testing was done both by the developer team and the client in an exploratory form, to find possible bugs and required other fixes – many were found and fixed. We did not use formal bug classification systems, but instead just fixed them in a way we saw appropriate.

Finally, we conducted a formal usability testing session with four of our end users with different backgrounds to validate our requirements and find usability issues. End users were tasked with performing operations such as signing in and booking rooms, while

using their own devices. Users were monitored and asked clarifying questions regarding usability as they performed tasks. Device screens were recorded with a camera to collect details such as how long operation take or what the participants do precisely. The results were summarized in a summary document and processed into refined requirements and usability fixes, implemented in the last two sprints.

2.2 Testing roles

During the project, developers were tasked with testing their own code against written user scenarios manually and write unit tests for their code. Since every developer tested their own code, the testing responsibility of the frontend went to frontend developers, and backend testing for backend developers. However, system testing responsibilities fell for the whole team, and partly for the customer as well. Developers also help in verifying each other's code via pull requests and related code review process.

Test automation also had an important role in testing the software. The CI/CD responsible had taken care of building the integration pipeline and the pipeline included style checks and regression tests for both frontend and backend.

The customer participated in testing by performing ad hoc usability, system, and acceptance testing. Usability testing was used to validate requirements in staging environment and customer acceptance was required before deploying the product in the production environment.

Last party involved in testing were the end users. End users participated in the usability test session, providing invaluable opinions on how the product should work, look, and feel. From these results, a list of usability fixes was assembled and partially implemented.

2.3 Test schedule

Table 2.1 presents the planned and actualized testing schedule for the project. The bolded sections in the actualized column highlight differences between planned and actualized testing.

Weeks	**Planned**	**Actualized**
35–42	Requirements gathering and validation. Requirements review meet with the client 6.9. UI design review meet with the client 22.9. Written scenarios 26.10.	As planned.
39–50	Unit testing.	Unit testing backend.
40–50		**Unit testing frontend.**
47	Ad hoc testing and acceptance for MVP.	Ad hoc testing
48	Usability test session. Plan fixes. Integration test implementations if any.	Usability test session, **acceptance for MVP features**. Planning usability fixes. **Manual system testing.**
49	System testing and reporting. Fixes.	As planned.

50	Last fixes. Final reporting.	As planned.

Table 2.1: Testing schedule summary.

As seen from the Table 2.1, unit testing the frontend and acceptance testing the MVP features started about a week later than planned for. However, we still managed to keep up with the schedule in terms of timing of the usability testing session. While there was some time left for system testing, there was not a time margin big enough for implementing automated integration tests. Thus, the system testing was implemented manually, as documented in section *4.4 System testing*.

2.4 Test documentation

The results of regression tests are updated on every merge with the CI/CD pipeline running the tests as a commit hook. The results can be found from the pipeline log in the Actions log in GitHub.

Bugs found from ad hoc testing and manual system testing, are less coherently documented, but the principle has been to log issues in the Trello board before fixing them, so our Trello should contain an up-to-date list of bugs and other issues, and their fix status.

For usability testing there are a couple of documents that were created for testing. There is a usability test plan document that was shared in the test plan. There is also a note template for the observer that wrote down the notes. The notes for each test candidate can be also found. Finally, there is a detailed report of the results, and a summarized report of the results can be found on Appendix A.

3 Testing Tools

For unit tests, we used Jest unit testing framework for both backend and frontend, and additionally react-test-library for testing the frontend. GitHub Actions was utilized in setting the tests to run on every push to the repository. The summary from the unit tests includes a count for passed, failed and all tests by frontend and backend separately, but apart from that, the group did not see value with further statistics compiling.

Like unit tests, for the system testing part, we stuck to the simplest form, which was to test manually features as they were integrated in the product, and log possible issues to Trello board.

For the usability test a camera was used to record the session together with a set of mics that were attached to the facilitator and the participant. There were some technical issues that caused some of the recordings to be cut short and not everything was filmed as planned. For the usability test session, the participants used their mobile phone to access the PWA. However, ¾ of the phones encountered technical issues with the test account and the participants had to use the facilitators or the observer's phone. The statistics observed can be found in Appendix A.

4 Test cases and results

4.1 Test results

Unit tests cover 30 cases in the frontend and 138 in the backend and 100 % of the cases pass at the moment of hand-over. Test automation is implemented for regression testing and can be extended for integration tests once some are implemented in further development. Manual system testing covered all implemented features.

Found deficits that are not yet fixed constitute from graceful handling of situations where there are no available rooms or fetching room data fails and frontend crashing if connection is lost to the server. We also recommend the customer to consider publishing terms of usage and a GDPR-compliant report of the handling of person data. Additionally, in Table 4.3 can be found a list of possible future usability fixes or further development ideas.

4.2 Unit testing

In the frontend, the repository had 30 unit test cases, divided in 7 test suites, by the end of the project. Out of the 30 test cases, 30 passed on the moment of handing over the project. The backend had 138 test cases in 19 test suites, out of which 100 % passed. Possible bugs found by the tests were fixed right away, so we leave 0 identified bugs for the future developers.

4.3 System testing

Table 4.1 lists the planned system test cases derived from integration test cases documented in test plan, and their pass statuses. Like we had anticipated, the group resources were enough only for manual system testing instead of implementing automatic integration tests. However, the team managed to get commendable test coverage for the whole system. Testing was achieved by both the development team and the customer performing tests.

Test case	Pass	Fail
Login		
Login with Google SSO succeeds with correct credentials	**X**	
Login fails due to wrong credentials	**X**	
First time Office selection		
Office location is asked on first login	**X**	
Available rooms are shown from the office location selected after first login	**X**	
All rooms are shown if office location selection is skipped in first login	**X**	
App bar		
Preferences screen can be accessed from the app bar	**X**	
Booking screen can be accessed from the app bar	**X**	
Signing out is possible from the app bar	**X**	
Logout		
Logout redirects to the login page		
Trying to go to other screens after logout redirects to login	**X**	
Room list		

Room listing shows the available rooms at the user's preferred office location	**X**	
Room listing shows the capacity of the rooms	**X**	
Rooms show remaining free time correctly when there is a booking coming for a room	**X**	
Room listing notifies the user when no rooms are available		**X**
Room listing notifies the user when room data cannot be fetched		**X**
All room features can be expanded with the switch	**X**	
Individual room features can be toggled visible	**X**	
Booking		
Available rooms can be booked for 30 or 60 minutes	**X**	
Available rooms can be booked only for 30 minutes if 60 is not possible		
Booking fails if the room was just reserved.	**X**	
Current booking		
The user can see their own current booking	**X**	
User sees all their current bookings if there are more than one	**X**	
The user can add more time to their current booking	**X**	
User cannot add more time than is left before next booking	**X**	
The booking is removed after pressing delete	**X**	
The booking is not removed if removing fails	**X**	
Notifications		
Notifications appear on successful and failing operations	**X**	
Error notifications stay visible for 30 seconds	**X**	
Notifications can be dismissed	**X**	

Table 4.1: Planned system test cases and their testing statuses.

As displayed in Table 4.2, most features of the software have been tested. The two not passing cases relate to implementing handling for situations when there are no available rooms or fetching room data fails.

Informal exploratory testing was also done by the development team as well as the customer. Findings from these testing sessions were reported in the common Slack platform at least from the customer's part, after which the development team fixes the bugs.

4.4 Special testing

The program's security was tested by creating a list of different cases and then by testing some of them in practice and checking that the program is as secure as it should be on those parts. For most of the other cases, a way to mitigate the security flaws were thought of and documented in a separate security report, included as Appendix C.

4.5 Acceptance testing

Acceptance testing was performed by the customer in the staging environment for each new feature continuously, before releasing the features in the production environment. The customer had their own timetable and methods for testing but delivered comments and approvements in a timely manner after release notes being posted on the common Slack channel.

4.6 Usability testing

For the MVP a usability test session was conducted to find out any usability issues or possible bugs in the system. There were 4 participants that tested the application on a mobile phone. The participants were given a set of 6 tasks which are in Table 4.2 with their success rates as well.

#	Task	Success rate
1	Onboarding process - Download, login and set user preferences on the application	33 %
2	Take a look at the main view and find out if there are currently available meeting rooms.	100 %
3	Book a meeting room	100 %
4	Add additional time on the current booking	100 %
5	Delete your current booking and book another room	75 %
6	Add additional time to the current booking. The additional time fails, does the participant realize why?	50 %

Table 4.2: Usability test tasks and success rate %

Tasks 2-4 had a success rate of 100 %. On task 5 one person failed the task due to lack of understanding how the interface works. This was a good finding because it showed that clarification needs to be made for better usability. Task 6 had a success rate of 50 %, which to some extend was also the result we were looking for since at that time the application was not informative enough in the UX designers' opinion. Task 1 success rate is at 33 %, but the findings for the onboarding process cannot be evaluated reliably. There were a lot of technical issues, since the participants could not use their own accounts to sign into the application and this created issues with their mobile phones. This meant that the participants could not use their own phone and had to borrow the observers or facilitators phone to test the application.

The results have been summarized further in Appendix A in Finnish. In Appendix A the two last columns are fixes that were done before returning the project to the client and possible future development ideas. The fixes that were done included:

- Being able to compare additional information between the rooms
- Adding a toggle button to toggle all room additional information or hide all room additional information
- How long the currently booked room is free for
- Capacity needs to be shown next to the room title
- Notify the user that they cannot book another room if they have an ongoing booking
- Hide buttons that cannot be used (60min booking button, +15min additional time button)
- Login page needs to be clearer and less pixelated with a better design
- Error messages should be closed by the user
- Remove italic font
- Remove collapse elements if they do not have additional information in them.

As mentioned earlier the last column on Appendix A contains information on possible future development information that is very valuable information for the client since they will be developing the product further. These ideas and their reasoning can be found below in Table 4.3 as well.

Further development idea	Reasoning
Information on where each meeting room is located within the office	There was a high need for being able to see where some meeting rooms were located within the office. This information would be especially important for employees that visit other offices.
Custom times for booking and additional time	Default times are not flexible and sometimes there is a need to put more time into a booking than what is offered on the application.
Custom available room order	The current order is alphabetical, but this was not logical to some participants. Order in capacity or order in physical appearances would be better.
Office preference should be in an easy reach place	The participants could not find the office preference setting.
Better visual branding for Vincit and Vincit humor in text	To the participants it did not feel like Vincit.
Accessibility	There is a blind worker at Vincit.
Need for a google play application	Some participants found it suspicious to load an application from a website.
Let the user book more than one room	Sometimes a user does not want to remove a current booking, but still be able to book other rooms.
Book other resources, e.g., the office car	There are plenty of other resources that can be booked on Google Calendar.
Search available rooms with a criteria	Some participants did not agree with the way the search was happening and wanted to be able to search with a specific criteria.
Book a pop-up workstation	These can also be booked on Google Calendar.
Room suggestions if time runs out from the booking and the room is booked to someone else, but the meeting has not ended.	Sometimes meetings get extended but the meeting rooms might be booked, and a new room needs to be found. A suggestion of possible rooms could aid the user.

Table 4.3: Further development ideas and the reason behind them.

With these results the research questions that were formed on the usability test plan were answered. The answers can be found on Appendix B in Finnish.

5 References

[1] Feature Branch, https://martinfowler.com/bliki/FeatureBranch.html

[2] Code review, https://smartbear.com/learn/code-review/what-is-code-review

[3] Prettier, https://prettier.io/

[4] ESLint, https://eslint.org/

APPENDIX A

Tehtävät	**Onnistumisprosentti**	**Kulutettu aika**	**Testaajien kokemat hankaluudet**	**Havainnot**	**Kehitysideat (mitä me toteutamme)**	**Jatkokehitysideat**
Tehtävä 1 – Onboarding (asennus, kirjautuminen, toimiston asettaminen)	33 %	Asennus: 20sek, Kirjautuminen: ~45sek Toimiston asettaminen: ~15sek	Testikäyttäjätunnukset Tarve päästä takaisin napilla vaihtamaan toimistoa.	Onboarding ei pystytty kunnolla testaamaan kirjautumishankaluuksien takia. Myös kirjautuminen täysin tuntemattomilla tunnuksilla vie huomattavasti enemmän aikaa kuin omat tutu tunnukset. 2/4 ei käyttänyt PWA:ta.	Parannellaan sisään kirjautumisnäkymää.	Tarve google play sovellukselle

Tehtävä 2 – Huoneiden tutkimista	100 %	-	Henkilömäärä voisi olla esillä suoraan. Collapse elementin auki klikkailu on turhan työlästä	Näyttää helppokäyttöiseltä Tietoa on riittävästi Terminologia on ymmärrettävää	Henkilömäärä saisi näkyä suoraan: ilman, että joutuu klikkailemaan auki. Toggle kaikkien huoneiden lisätietojen näyttämiselle/piilottamiselle kerralla	Mistä huoneet löytyvät toimistolta? mm. jonkinnäköinen kartta tai vastaava tieto tarvitaan.
Tehtävä 3 – Huoneen varaaminen	100 %	32sek, 45sek ja 119sek. Itse varaus 2 painallusta.	Huone lisätietojen kliksuttelu auki aiheutti turhautumista.	Alle 30min ajanvaraukselle ei tarvetta Yleensä 60min sopiva, voi esiintyä tarve 45min tai +60min. Hyvä huomioida, että osa piti ideasta piilottaa huoneen lisätiedot.	Pitäisi voida vertailla huoneita niin, ettei yhden huoneen klikkaaminen auki piilota muita. 60min nappi piilotetaan, jos ei voi tehdä 60min varausta.	Haetaan ensin, kuinka pitkäksi ajaksi huonetta tarvitaan, ja millä varusteilla. Tämän jälkeen tulisi tähän kriteriaan sopivat huoneet. Custom ajan varaus – varataan huone jäljellä olevalla ajalla esim. 36min tai 2 h
Tehtävä 4 – Huoneeseen lisäaikaa	100 %	~5sek	Mitä jos en haluu +15min, lisäaikaa. Mitä jos haluaa tunnin?	Intuitiivista, nopeata ja helppoa		Custom lisäaika (varataan jäljellä oleva aika). Sovellus ilmoittaa, kun varaus on päättymässä.

Tehtävä 5 – Huoneen poistaminen ja uuden huoneen varaaminen	75 %	~10sek	Yhdelle testaajalle ei ollut intuitiivista poistaa vanha varaus, ja ei ymmärtänyt miksi ei uutta varausta voi tehdä. Valittu huone ”Stand up meeting point” aiheutti kaikille ihmetystä missä huone sijaitsee. Osa ei kokenut huoneiden aakkosjärjestystä loogiseksi.	Suurin osa testaajista poistivat varauksen ennen kuin varasivat uuden. Itse poistaminen helppoa. Yksi testaaja ei ollut varma haluaisiko oikeassa tilanteessa poistaa varausta. Erilaisia ehdotuksia huone järjestyksille: Aakkosjärjestys, henkilömäärä-järjestys, huoneiden alueittainen järjestys. Tässä painottui tarve tietää missä huoneet sijaitsevat.	Selkeä ilmoitus, että uutta huonetta ei voida varata, jos on jo varaus olemassa.	Sallitaan useampi varaus käyttäjille. Custom huonejärjestys. Jos kyseisessä huoneessa loppuu aika ja kyseiseen huoneeseen on jo toinen varaus tulossa, voisi sovellus ehdottaa toista samanlaista neukkaria, joka on vapaana vielä.
Tehtävä 6 – Lisäajan laittaminen ja sen epäonnistuminen	50 %		Notifikaatio katoaa liian nopeasti Ei oltu varmoja miksi lisäajan laittaminen epäonnistui	Olisi tarve nähdä tämänhetkisen varauksen mahdollinen tuleva varaus. Tarpeellista kertoa, miksi lisäaikaa ei voitu laittaa.	Virheilmoitus pidempään näkyvillä. Näytetään tämänhetkisen varauksen tuleva varaus. Piilotetaan tai disabloidaan	

					+15min lisäaika nappi	
Yleisesti	-	-	Epäselvää, mistä toimistopreferenssit voidaan vaihtaa. Huoneita hankala vertailla Sovellus ei ole saavutettava	Kaikki kokivat, että sovellus on helppokäyttöinen Toimintalogiikka selkeä Pienien muutoksien kera, kaikki käyttäisivät sovellusta. (huoneiden sijainti, custom ajat) Toive muiden resurssien varaamisesta Tapahtumakohtaisesti käyttäisivät Google Calendar vs. Get A Room.	Kursiivisuus pois	Toimistopreferenssien siirtäminen jonnekin intuitiivisempaan paikkaan. Saavutettavuus pitäisi ottaa huomioon, jotta kaikki Vincit työntekijät voivat käyttää sovellusta. Popup työpisteiden varaaminen. Muiden resurssien varaaminen Vahvempi visuaalinen Vincit brändäys ja Vincit huumoria Favourite neukkarit.

APPENDIX B

Tutkimuskysymys	Vastaukset
Tuottaako sovelluksen asentaminen ja kirjautuminen ongelmia – onboarding process?	• Asentaminen ei tuottanut ongelmia. Kuitenkin nettisivulta asentaminen vaikuttaa epäluotettavalta. • Muuten ei tähän pystytä luotettavasti vastaamaan.
Onko Päänäkymä ymmärrettävä	• Päänäkymässä käytetty terminologia on ymmärrettävä • Aakkosjärjestys ei kaikille ollut looginen – jotkut suosivat sijaintijärjestystä tai henkilömääräjärjestystä. • Toimiston vaihtaminen tuntui vaikealta / ei loogiselta

Miten nopeasti osallistuja pystyy varaamaan itsellensä huoneen?	• Varaaminen oli nopeaa ja helppoa. Sopivan huoneen löytäminen ominaisuuksien perusteella oli kuitenkin työläämpää. • Varauksen saa tehtyä 2 painalluksella – sopivan huoneen löytäminen lisää painalluksia • Yksi virhetilanne, kun testaaja yritti varata huonetta ja epäonnistui, vaikka huone oli vapaana. Ei saatu toistumaan. • 30|60min vaikutti sopivalta ajalta, mutta custom ajat voisivat olla paremmat. Ei kuitenkaan tarvetta alle 30min varauksille.
Miten nopeasti osallistuja pystyy muokkaamaan omaa varausta (lisäaika + poistaminen)?	• Poistaminen oma-aloitteista ja helppoa • Lisäajan laittaminen oma-aloitteista ja helppoa • +15min lisäajan tilalle toivottiin custom aikaa.
Löytyykö selkeitä käytettävyysongelmia?	• Epäselvää, että vain yksi huone voidaan varata kerralla – käyttäjä tarvitsee lisäohjeistusta • Epäselvää, miksi virhetilanteita syntyy esim. huoneen lisäajan laittamisen epäonnistuminen – käyttäjä tarvitsee lisätietoa. • Ajallisesti huoneiden varaaminen nopeaa ja helppoa – ei löytynyt selkeitä ongelmia • Käyttäjä ei pysty tekemään varausta hänen toivomallaan tavalla. Custom aikaa toivotaan. Myös käyttäjällä toive löytää vapaita huoneita kriterian avulla, esim. ensin annetaan aika ja ominaisuudet ja vasta sitten näytetään vapaat huoneet.
Tuleeko osallistujalta kysymyksiä testauksen yhteydessä?	• Lisäajan epäonnistumiseen tuli kysymyksiä – ei ymmärretty miksi lisäajan laitto epäonnistui • Mistä vaihdetaan toimistopreferenssit? • Näkeekö missä neukkarit löytyvät? • Näkeekö että kuka on varannut neukkarin? • Miksi quick book ei toimi? (kun on jo varaus) • Onko 60 min maksimi? • Miksi lisätiedot ei näy suoraan? • Eikö pysty varaa tulevaisuuteen? • Miksi on rajattu varausaika? • Ilmoittaako tämä, kun varauksen aika on lopussa? • Pystyykö sokea käyttämään?
Mitä mieltä osallistuja on sovelluksesta?	• Helppokäyttöinen • Logiikka vaikuttaa hyvältä – kuitenkin pari parannusta tarvitsisi, jotta kaikki käyttäisi kyseistä sovellusta. • Kts. Jatkokehitysideat.
Jos osallistuja vertailee sovellusta google kalenteriin, miltä käytettävyys tuntui siihen verrattuna?	• Preferenssi Get a Room! Vs. Google Calendar – miksi? o Get a Room – ellei tarvitse kutsulinkkiä silloin GC

	○ Tapauskohtaisesti – ad hoc/matkustaessa/kone on kiinni = Get A Room

APPENDIX C

Introduction

This document aims to identify and give solutions to the possible issues and vulnerabilities related to the security of *Get a Room!* software project.

Even though some potential problems were identified, the software can be considered secure. It doesn't contain any evident or known vulnerabilities that would allow unauthorized personnel to enable access to the software or data. The software is also intended to be used internally in an organization, so it won't probably be that well known and thus will likely not face many people trying to take advantage of it.

Issues

As mentioned in the introduction, there aren't many vulnerabilities that could be considered plausible to exploit. However, some possible issues are listed below in two different categories, user-related and technical.

User-related issues

User-related issues are related to the user doing something incorrectly or negligently, causing a third party to gain access to things they should not have. Overall, user-related issues can be considered a much higher risk for the software's information security than technical issues.

Simple and guessable passwords

One of the most common user-related issues with this software could be passwords that are too simple or easily guessable by bots. It is likely that at least at some point, a bot comes by the software and tries to guess users' passwords through the Google login. This issue is easily mitigated by making the users select hard-to-guess passwords instead of simple ones.

Social hacking

Social hacking could be considered a likely issue. Social hacking could be done in multiple ways, but the most common way is to get somehow the user to give the email and password to the hacker.

One way to mitigate this issue is to educate the users and force everyone in the organization to enable 2-factor authentication. As in this case, at least most users will be IT professionals, so it can be expected that the users are well educated on this matter. Overall, the chance of social hacking in this application can be considered low, and the damage it can cause is minimal.

Stealing the user's token

After a user has logged in to the application, a JSON web token (JWT) containing user identifying information and two different tokens issued by Google is placed into the browser's cookies. As the JWT is unencrypted and has all the data required to impersonate the user to Google's service, this kind of attack would be viable.

The JWT is stored as an http-only cookie to mitigate this issue, which means that only the server can read the cookie's content programmatically. This prevents any malicious JavaScript in the user's browser or computer from reading the cookie's content. Because of this, the only way a hacker would get access to the JWT is by getting the user to send it to them or by having remote control of the user's computer, which would have required the user to permit them to do so.

In the future, to make it substantially more complicated for a bad actor to steal JWT's content, the content could be encrypted on the server before setting the cookie's content. This way, even if the hacker would get the cookie's content, they would not be able to use it before decrypting.

Technical issues

Technical issues are mostly vulnerabilities that some actors can exploit to get access to user data or, in the worst case, have access to the servers running the software.

Vulnerability inside one of the dependencies

As the software has a large number of different dependencies, it is highly likely that at some point, some of those have vulnerabilities. In the worst-case scenario, this would mean that an attacker could utilize a vulnerability of one of our dependencies and get access to the server or user data.

The issue is mitigated by only using reputable and popular dependencies that are updated regularly and that will most probably receive a fix for the vulnerability as soon as possible. In addition, the number of dependencies should also be kept to a minimum to minimize the risks of such vulnerabilities.

Man-in-the-middle attack

In a successful man-in-the-middle attack, an attacker would read the incoming data from both sides and control what gets sent where. By using this method, the attacker could possibly be able to receive the user data that the software has access to.

The issue can be mitigated by using popular and well-known web application frameworks that have the latest security-related settings set correctly by default and that are updated regularly with the latest security updates.

Highly unlikely vulnerabilities

There are, of course, many highly advanced vulnerabilities that can affect this software and that can't be prepared for at our scale. Still, the chance that these would be used against our software is virtually non-existing as the attacker would have very little to gain by getting access to the software's data compared to the vast amount of resources it would take to orchestrate such an attack. Examples of such vulnerability could be an attacker exploiting a vulnerability in Google SSO or, for example, a zero-day vulnerability for any of the platforms used.