Final Report

1. Introduction

1.1 Purpose

This document describes the approaches used to achieve the goals in each phase during the project, the activities performed, the results of the activities and personal reflections on the project and this entire training course.

1.2 Scope

The content of this document focuses on the approaches used, the processes determined, and the findings and retrospectives carried out in the activities. For details of activities or results, refer to the Software Design document and the Security Assessment Report document.

1.3 Definitions, acronyms, and abbreviations

1.4 Reference

- Software Requirements.pdf (team4)
- Software Design.pdf (team4)
- Software Requirement Specification.docx
- Security Requirements.xlsx
- https://owasp.org/www-project-top-ten/

- Asset repository: https://github.com/hijang/lsc_cctv

2. Phase 1 - Development

2.1 Overview

We have developed a secure implementation of the system as required by the project description.

2.2 Plan

2.2.1 Approach

Strategies to achieve the quality goal of project are as follows:

- Identifying quality attributes and making a design decision to select the optimal alternative among the alternatives to satisfy the attribute.
- Identify threats by exploring possible threats as much as possible.
- Among the identified threats, a high-priority threat is selected to derive mitigation and implement it.
- Make secure code during implementation to prevent threats that may occur due to internal vulnerabilities.

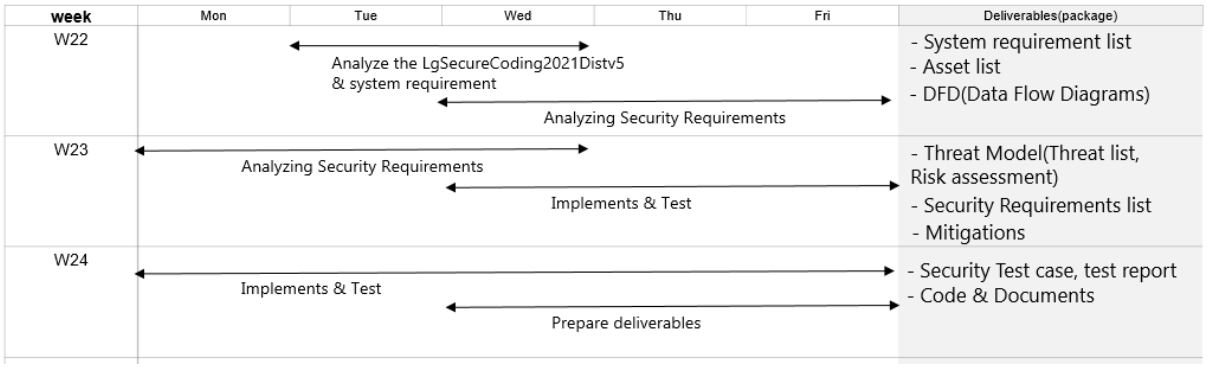
2.2.2 Team building & scheduling

In the beginning, the rough roles and schedule plans were divided as follows.

Initial R&R

Lead & schedule management: Daesik Kim
Architect: Kyungsik Lee
Configuration management & build: Hanil Jang, Yujin Lee (use github)
Acceptance Test & QA: Dahee Jung, Jinchul Kim
Risk Assessment & Mitigations: All
Development & Documentation:
- Application(User Interface) side: Hanil Jang, Yujin Lee, Dahee Jung
- Jetson Nano side: Daesik Kim, Kyungsik Lee, Jinchul Kim

Initial schedule



It was possible to divide roles according to the initial R&R until base code analysis and initial context analysis, but the division into application side and server side soon became meaningless.
In the end, the work items were distributed in a way that assigned the identified work items to team members, and the progress of the work was shared every day.
We found that it is better to identify, divide, and share small work items rather than dividing roles by function or area. So, in phase 2, we decided not to divide roles by area, but to create a backlog with identified work items, share daily progress, and decide what to do next.

2.3 Defining a system purpose, objectives, requirements

It was decided to apply the security attribute first. Therefore, it is defined by minimizing the functionality so that the functionality can be focused on the security properties rather than the high functionality.

Using ACLs to enforce separation of privileges and least privilege was a good decision. However, you should have considered the situation where 2-factor authentication is needed.

2.4 Identifying security asset

Assets had to be identified to derive and assess threats. Types of assets include electronic information, software, and hardware (networks, servers, personal computers), which are used to identify assets. In particular, assets that need protection in terms of confidentiality, integrity and availability were classified as security assets, and this phase determined whether the assets needed protection through threat modeling that could arise from each security asset.

After reviewing the asset assessment of Team 5, in the previous phase1, our team did not identify each application and the learning data of the application as a secure asset. We were able to learn once again that asset identification was needed so that even the software and electronic information parts were not omitted.

2.5 Identify threat

We performed threat modeling to identify threats that can be exposed during the lifecycle of the Tartan monitoring system.
The candidates for the treats modeling approach considered are as follows.

- STRIDE
- Security Cards
- PnG

Among them, it was decided to apply STRIDE and PnG together. This is because STRIDE had the most experience and proficiency among each method, had the most available tools, and was expected to identify the most possible threats. In addition, PnG was additionally

applied because it was expected that it would be helpful in additionally identifying usable threats from the attacker's point of view.

It is regrettable that the threat analysis of the entire certificate management life cycle was insufficient because it was not considered a dynamic perspective.

2.6 Selecting threats to mitigate and defining security requirements

It is impossible to deal with all identified threats due to project constraints, so it is necessary to analyze the threats that have a great impact on achieving the security goals of the Tartan monitoring system and deal with the high-priority threats.

Consideration: evaluation methodology

Prioritization requires an assessment of impact and likelihood. The evaluation of impact and likelihood is inevitably influenced by the subjectivity of the evaluator, and a method to supplement the maturity of evaluation within the project is needed.

Decision: OWASP Risk Rating Methodology

OWASP Risk Rating Methodology includes a scale that can be used for evaluation, and a framework for severity evaluation is formed, so it was decided to use it for evaluation.

The risk evaluation determines the threat to be mitigated, and then derives security requirements for mitigating these threats.

Although the risk rating method was used, the subjectivity of the evaluator appeared to have a significant influence on the evaluation of the severity of risk. There was a deviation in the evaluation of individuals, so it had to be decided through risk rating review after reviewing and agreeing with all team members.

2.7 Design mitigation

The workshop, where the entire team participated, decided how to mitigate the selected threat. The alternatives, decisions, and reasons reviewed by design workshops are set out in the 5.Design Decisions section of the Software Design document.

There was a failure to consider mitigation of the initial installation of Windows' certificate because it did not take into account the entire life cycle of the asset. This lack of dynamic perspective analysis seems to be the cause.

Among the mitigation methods selected in the initial design workshop, there was a compromise during the implementation process by performing different actions than expected.

2.8 Implementation (secure coding)

To keep the code secure, perform the following activities.

- For each pull-request, perform the code review by referring to the C++ secure coding guide of the LG Electronics Security Implementation Guide.
- In the integration stage, review analysis results of the tools below.
    - code X-ray: Review for severity greater than Major
    - sonar cloud: Review for severity greater than Major
    - FlawFinder: Review for severity greater than level 2

As described above, we planned to process code review for every pull-request. However, as time went on, it was skipped for convenience. Urgent patches which were not verified enough were frequently updated whenever fixes were needed.

Since we did not set up an automation test to prevent regression issues, it was difficult to verify the integrity of each integration version. As a result, the version that was working normally also malfunctioned after updating several patches, and it was necessary to make an effort to fix it.

There were candidates for static analysis tools to use and SonarCloud is one of them. To apply it, we need to set up a build environment. The build environment was ready on the target board(a.k.a Jetson Nano) which is an ARM device but, Since SonarCloud only supports x86 environment, we could not go further.

2.9 Verification

Test cases were used to verify security design and implementation. A total of 19 test cases were written focusing on security requirements.
The test used the scenario test technique and focused on verifying the application of mitigation measures to the identified security threats.
For verification, system test level tests were performed, and unit tests and integration tests were not performed.

The test was conducted in such a way that one tester performed the test case before the software release and reported the result. Before the final release, the pass of 19 test cases was checked and released.

We did not consider the frequency and conditions of conducting the test in detail.
There was no explicit definition of what level of testing to be conducted at what time, and the test performance cycle was long. There was a problem that the installation guide document was missing essential installation content because the final documents were not verified.

The parts that we think need to be supplemented in the verification activities are as follows.

- Vulnerability analysis and verification of the environment and infrastructure in which the implemented module runs
- Identification of potential security vulnerabilities through the use of security analysis tools such as fuzzing tools and penetration testing tools

- Automated code fetching, review, static analysis, deployment and vulnerability identification pipeline creation through application of DevSecOps
- Establishing a specific test plan and securing the reliability of the verification process through this
- Verification of correspondence between installation and usage guide documents and actual product operation

3. Phase 2 - Evaluation

3.1 Approach

3.1.1 Assessment Objectives

Find vulnerabilities to interfere with achieving security goals of target system

- Confidentiality - Vulnerabilities to leak video stream data or stored personal data in the system
- Integrity - Vulnerabilities to tamper or delete Video stream data and logs
- Availability - Vulnerabilities in which to stop the service or inhibit quality of service
- Non-repudiation - Vulnerabilities to avoid the usage history of the service and records of access

3.1.2 Strategies

Identify vulnerabilities in the target system with the following strategy:

- Finds that there is a common vulnerability that may occur.
- Identify the vulnerability to the subsystem used by the target system.
- Investigate mitigation for security assets and find weaknesses of mitigations.
- Find mistakes or flaws in Design or Implementation.

3.1.3 Tactics

For each strategy, perform the following tactics.

- Find vulnerabilities that correspond to OWASP top 10.
- Do fuzz testing in order to find vulnerabilities about input validation.
- Find whether there are vulnerabilities in Jetson nano board and configuration or not.
- Review the result of static analysis in order to find vulnerabilities in code.
- Review design documents and security requirement documents to find out if there is anything missing.
- Review the code for looking for factors that can cause vulnerabilities.

3.2 Plan

To efficiently carry out the above tactics within the project deadline, we decided to divide work items and assign them to team members.

3.2.1 Work items

A list of tasks required to complete 2.3 tactics was identified and assigned through the planning workshop. In fact, the content of Tactic is derived from the planning workshop.

Key work items identified are as follows.

- Investigate vulnerabilities corresponding to OWASP top 10 vulnerabilities
- Fuzz testing

- Penetration testing using Kali, etc.
- Subcomponent, vulnerability investigation on underlying systems (linux)
- Static analysis
- Mitigations review
- Test case review
- Create exploit for discovered vulnerabilities
- Mitigation Suggestions for Vulnerabilities Found

Priority of work items

Since our first goal is to find and report vulnerabilities in the target system, we decided to proceed with the task of finding vulnerabilities first, and making suggestions for mitigation of the discovered vulnerabilities and creating exploits for the vulnerabilities was decided with a second priority.

3.2.2 Monitoring & control

The project has a set deadline, and it's not enough. In order to achieve as many goals as possible, we decided to hold a daily meeting such as a daily scrum every afternoon to ensure transparency on the progress of work items.

For each work item, time boxing of 1 day was set, and it was decided whether to proceed further or to stop at the daily meeting.

Investigation was carried out for each work item, and there was hardly any work that exceeded the set deadline, but fuzz testing required a lot of work to prepare for the oracle and test runner, and the test took a long time, so much effort was consumed than initially expected. It may have been a wise decision to switch to another task early, but since there was a lot of preparation in the beginning and it was an interesting task, we decided to put more effort into finding the vulnerability.

3.2.3 Team building

Writing documents and preparing presentation materials were shared by everyone.
The other identified work items were divided as mentioned above.
Roughly, the kind of work each team member did is as follows.

- Artifacts and code review: Hanil
- Penetration testing: Daesik
- Test case analysis: Dahee
- Static Analysis: Jinchul
- Input validation/testing setup: Yujin and Kyungsik

3.3 Fuzz testing

We tried to discover vulnerabilities through fuzzing on the module with user input of the sfid system and the network port opened to receive traffic from the external network.
(For more details about fuzzing, see the Security Assessment Report, 4.1 Fuzz testing)

Fuzzing target

We tried to perform fuzzing on a module that has input values from the outside for each of the client and server, and as a result of the system analysis, the following targets were identified.

- IP input value in client app
- Person name input value in client app
- Port for receiving data from clients in the server

Fuzzing strategy

For each input of server and client, fuzzing strategy was established as follows.

- Inputs in client app :
    - The client app is a QT-based GUI app, and as a result of the survey, there is no applicable QT-based GUI fuzzing tool.
    - To use the existing CLI based fuzzing tools, create a harness that tests the API that processes user input in the client app.
    - Perform fuzzing on the created harness.

- Inputs in server :
    - Identifies the port for the server to send and receive control and data with the client.
    - Attempts to connect to the identified port.
    - Send random data to the connected port.

Fuzzing tools

We analyzed the tools used for the purging of the sfid system with the following five evaluation criteria.

Fuzzing tools	QT GUI app fuzzing support	Network payload transport support	Fuzzing type	Previous experience
AFL	Not supported	Not supported	mutation based dumb fuzzer	Experienced
Peach Fuzzer	Not supported	Not supported	smart or dumb fuzzer	Not experienced
WinAFL	Not supported	Not supported	mutation based dumb fuzzer	Not experienced

sfuzz	Not supported	Supported	dumb	Not experienced

As a result of the survey, there were a number of various fuzzing tools, but to perform fuzzing within a limited time, the applicability was reviewed mainly with familiar tools with previous experience, and the following two fuzzing tools were selected.

- AFL (IP input value in client app, Person name input value in client app)
- sfuzz (Port for receiving data from clients in the server)

Additional work for fuzzing

The sfid system is a server-client based system, and when the server crashed during fuzzing, there was a problem that the server was not recovered to its normal state. This made it difficult to perform continuous fuzzing. So, when the server goes down, we created a code modification and a script so that it can be restarted in a normal state.

```
@@ -643,7 +649,7 @@ int main(int argc, char *argv[])
       if ((TlsConnectedPort = AcceptTcpConnection(TlsListenPort, &tls_cli_addr,
&tls_clilen)) == NULL) {
               conn_log.err("AcceptTcpConnection Failed\n");
               printf("AcceptTcpConnection Failed\n");
+               return(-1);
```

Reflection

- We learned that fuzzing is useful for automating the validation of random inputs of a specific module or function in a more robust manner.
- In the case of software based on GUI input, we learned that fuzzing is possible by creating a harness that uses the api of fuzzing target's specific library (so, dll).
- We confirmed that Team5's input validation using regular expressions was valid, and we learned that using regular expressions appropriately is important to prevent vulnerabilities.
- Given source code of the system was not managed by a git-like system so we are limited to give more details regarding security code analysis such as which code belongs to security development.

3.4 Penetration testing

Try to analyze and assess the target system, gather information and try to find vulnerabilities and exploit it. Sfid system consists of a client program running on Windows and a server program running on ubuntu.

The Below library and service should be installed to the operating system and that are mentioned in the Installation guide.

- Opencv, openssl, cryptmount, cryptsetup

Vulnerability analysis is performed sequentially for the following:

- Analyze data sent and received by servers and clients
- Analysis of libraries used/installed by servers and clients for operations
- Analyze vulnerabilities to other services, ports operating in a system/OS-enabled environment

	Analysis step	Tools / Method
Data analysis	It checks the network packets sent and received by the server and client, and checks whether important information is exposed or if authentication is underway between each other.	Wireshark, netcat
Library analysis	It checks the version of the library that the system must be installed to operate or required to operate. It checks the official page to see if there are any vulnerabilities in the library.	Internet search
Service and port analysis	It identifies which ports are being provided for which services in the environment in which the program operates. Verify that these services have vulnerabilities and that they can also affect the functionality of this program.	Netcat, netstat

After conducting each analysis, we will conduct the exit test for the relevant analysis.

(For more details about Penetration testing, see the Security Assessment Report, 4.2 Penetration testing)

Reflection

In the process of gathering and analyzing information about the OS, programs, and libraries for penetration testing, additional parts were found that were not considered for security at the time of phase1. It took more time to assess how the vulnerability could be exposed through the found vulnerability and what impact the vulnerability could give to the system/program than the process of finding the vulnerability.

3.5 Static Analysis

We reviewed all issues and tried to find vulnerabilities that can be the actual target of attack. Since the client does not have valuable assets, we focused on the server side. If we are lucky, we may be able to run a shell through code injection on the server.

(For more details about static analysis, see the Security Assessment Report, 4.3 Static Analysis)

Select static analysis tool to use.

Due to time limitations, it is not possible to get static analysis issues from various tools. Therefore, only one tool has been chosen to use. As the goals described above, we selected FlawFinder based on experience of use for the previous phase.

Review issues and choose issues to focus

46 issues from server side and 12 issues from client side have been detected by the tool. Among them, issues with potential for intrusion were selected first, and a total of 20 issues were selected. The client did not seem to have any merit in the attack, so it was excluded from the target, and 12 issues that occurred in the server were reviewed.

Compare issues for base code

As the advice of team mentor Clifford, we compare the issues of the base source code. The result shows that most of the selected issues which seems to be a vulnerability came from the codes written by the dev-team rather than base code. Even though several issues were still detected from base code, they had been already classified as ‘ignored’ on 1st review.

In-depth review for each issue.

The code in which the issue occurred was carefully reviewed to see if it could lead to an actual attack. I wanted to overflow a fixed-size buffer in the stack, but all of the fixed-size buffers defined locally did boundaries check which was brilliant. The issue we found a possibility for attack is overflow on global variable ‘`nameToRegister`’ which was detected as issue #19 by FlawFinder. There is code for boundary checking, but the logical operator was written incorrectly, and that could be a pretext for an attack.

Exploit using python script.

Python is good to use for manipulating the sending data and easy to connect, rather than the actual client app which needs to be built for every modification and trials. By debugging the server app, buffer overrun to the global variable ‘nameToRegister’ can be observed.

Retrospective

As we found an issue above, even there is an input validation code on the client side, but it should apply to both, as it can directly access the server to transmit maliciously generated data.

Even though we could not make a code injection through buffer overflow as we were aiming for, all steps we have followed was a good experience. It could be confirmed that an issue that was detected by the tool, but ignored, could be a reason for the exploit.

3.6 Mitigations Review

We evaluate what and how team5 decided to protect and review whether their mitigations are appropriate. Also we assess whether the commonly occurring vulnerability was present in the target system and their mitigations sufficiently prevented it.

Security asset, Mitigations, Security Requirements Review

We reviewed and compared team5's decision with ours. We mitigated it because we thought it was important, but we focused on comparing and investigating whether there were contents that were not taken.
In addition, we investigated whether there were any deficiencies by comparing the mitigate method.

By comparing and investigating the mitigated contents, it was very easy to find related vulnerabilities. I thought that vulnerabilities could be easily exposed if similar contents were not mitigated to the best practice level.

OWASP Top 10 Vulnerabilities

In order to find possible vulnerabilities, we refer to a list of commonly existing vulnerabilities to determine whether they exist.

A list of common vulnerabilities was to use OWASP Top 10 Vulnerabilities. This is because not too many of the known vulnerabilities lists are being used based on authoritative statistics. OWASP top 10 is a statistic for web applications, but it is expected to be fully tolerable because it is a service in a network environment.

(For more details about this, see the Security Assessment Report, 4.4.2)

Here, we review requirements documents to find relevance to the target system and investigate whether there are any associated API calls or associated function calls in source code.

Two possible vulnerabilities were identified by investigating the presence of the OWASP Top 10 Vulnerabilities. One of these (A9:2017) was discovered in other vulnerability analysis activities,
Only one(A10:2017) was discovered in this activity.

The results were below expectations as a result of the investigation with about 1 man-day.

In the case of non-Web applications, only about half of the items are applied, and many overlap with other approaches.

3.7 Test case analysis

If it was a completely unknown system, the situation could have been different, but phase1's experience of developing the same environment and starting reviewing artifact documents around test cases helped to quickly understand what the five teams implemented and what mitigation was.

The effort for this activity took longer than expected, including four days to check documents, due to problems setting up equipment environments for testing.

As a result, we could find one situation that was vulnerable.

Although the results did not come out well compared to the efforts, I think it is a must-have part of other teams' functional tests.

4. Overall Lessons & Learned

4.1. For project

- By comparing and investigating the mitigated contents, it was very easy to find related vulnerabilities. I thought that vulnerabilities could be easily exposed if similar contents were not mitigated to the best practice level.
- We thought that libssl-dev v1.1.1 installed by default in Ubuntu 18.04 would be the latest version, so we did not think to check the vulnerability. I had to use the latest version, v1.1.1k, which all vulnerabilities have been fixed.
- Thanks for its excellent endurance even when all team members log in and run the build at the same time. It's been a bit slow, but that was fine. Bye now, Jetson nano.

4.2. Personal Reflection

- Do not ignore static analysis issues, it may come with critical vulnerability.
- The same vulnerability also seems to vary in severity depending on the use scenario. While it is important to determine and apply threats and security measures, it is also important to define clear product functional requirements, usage scenarios, and operating environments.
- As it is very difficult to visualize the completeness of the system while analyzing the test cases, I felt that establishing and agreeing on metrics and criteria is very important.
- It's important to provide well-organized documentation from a security standpoint for the product's specifications, installation instructions, and usage guides.
- The fuzzing test can be useful for tests based on a large number of random inputs that are difficult for humans to do, but it is important for humans to well define the seeds of the input values in order to produce more meaningful test results.
- Even well-known open sources may have vulnerabilities, so it is important to have a habit of checking whether vulnerabilities exist before using them.