Felucca

MITS CMU

Statement of Work

Project Felucca

Team BugHunter

Authors

Di Mu (dimu@andrew.cmu.edu)
Sudi Lyu (sudil@andrew.cmu.edu)
Zihao Zhou (zihaozho@andrew.cmu.edu)
Guancheng Li (guanchel@andrew.cmu.edu)

Revision History

Version	Date	Change	Updated by	Reviewed by
0.1	Feb 7th, 2020	Structure of the document	Sudi Lyu	
0.2	Feb 10th, 2020	Scope of work	Guancheng Li	
0.3	Feb 11th, 2020	Overview and goals	Di Mu	
0.4	Feb 12th, 2020	Add module diagram and time sequence diagram	Sudi Lyu	
0.5	Feb 17th, 2020	Add requirements	Sudi Lyu Guancheng Li	
1.0	Feb 23rd, 2020	Update requirements Add plan and schedule	Sudi Lyu Di Mu Zihao Zhou Guancheng Li	Hasan Yasar
1.1	Apr 3rd, 2020	Add Pharos Knowledge	Sudi Lyu	
1.2	May 1st, 2020	Re-organize the contents	Guancheng Li Di Mu	

Contents

1 Introduction

1.1 Project Overview

The CERT Executable Code Analysis team develops and maintains a set of program analysis tools, known as Pharos that are based on an increasingly complex infrastructure. There are many tools in **the Pharos suite** (see https://insights.sei.cmu.edu/sei_blog/2017/08/pharosbinary-static-analysis-tools-released-on-github.html)and they perform a variety of functions to automate and support program analysis and reverse engineering tasks. Pharos tools typically export results in rudimentary data formats that are then ingested by other analytical tools. Despite generally using the same underlying libraries and infrastructure, the Pharos toolset is not easy to operate, inflexible, and individual tools do not integrate well together or with external tools.

1.2 Goal

The goal of this project is to develop a **Control Center** to unify the pharos toolset and make it easier to manage. This control center will enable program analysts to run, monitor, and export results from Pharos tools using a simple interface. Using the control center, users can submit jobs and retrieve the outputs after execution. Complex jobs other than single and simple executions are also supported. For example, multiple tasks can be executed in order. Furthermore, the output of execution can be the input of the next execution inside a job, whose pattern is defined by the user. In other words, complex jobs with multi-execution and pipelined-execution are supported.
The ultimate vision would be to develop a project-based model where each tool can be applied to a common set of artifacts as needed, the state is preserved as tools are run, and the results are easy to review and consume. Finally, we want to support different deployment models to adapt to different running environments.

1.3 Introduction of Pharos Tools

1.3.1 Overview

- **Environment**: The pharos tools are well dockerized and could run inside docker containers on any operating system.
- **Target**: The main target of pharos tools are 32-bit windows executable whose suffix is ".exe". However, pharos could also be used to analyze UNIX executable or 64-bit executable.
- **Test**: Pharos provides a test program with ".exe" suffix inside the "/test" directory.

1.3.2 Analyzing Tools

APIAnalyer

- Used to detect if target API is been called in this program
- The execution time is relatively long.
- Verbose logging level 1-14 (default 3)
- Inputs
  - Executable
  - Signature file in JSON format (default windows system API)
- Outputs
  - Output file in text or JSON format (not working)
  - Graphviz file in text format

OOAnalyer

- Used to analysis and recovery of object-oriented constructs program
- The execution time is relatively long.
- Input executable should be written in C++
- Inputs: Executable
- Outputs
  - Output file in JSON format
  - Prolog facts in text format
  - Prolog results in text format

PathAnalyzer

- Used to find paths between start address to goal address
- The execution time is relatively long.
- Inputs
  - Executable
  - Start address (Optional)
  - Goal address (Optional)
- Outputs: Z3 file in text format

CallAnalyzer

- Used for reporting the static parameters to API calls in a binary program
- Execution time is relatively short.
- Inputs
  - Executable
  - The file contains call list (Optional)
- Outputs: Output file in JSON format

FN2Yara

- Used for generating YARA signatures of functions, could be used to compare the similarity of programs
- Execution time is almost instant
- Inputs: Executable

- Outputs: YARA file in text format

FN2Hash

- Used for generating hashes of functions, could be used to compare the similarity of programs
- Execution time is almost instant
- Inputs: Executable
- Outputs: Hash file in JSON format

2 Requirements

2.1 Users

- **Analysts**: They get a deliverable that contains the Felucca control center and Pharos Kernel by an offline medium (U disk). Then, they use the felucca to execute Pharos tools analyzing binary executable on their machine in a completely offline environment because they concern a lot about security. All the execution would be handled locally and the output is stored in a local database. Generally, users can update Pharos tools by downloading the latest version and put it on the right path. Since there may be no internet connection on their machine, the kernel update also needs to be done through an offline medium. If they have a local network that could access the latest Pharos tools, they could just update it in an online manner.
- **Researchers**: They have less concern about security, so they could use the Felucca control server using a web browser. They do not need any pre-install components on their computer. Their executable will be uploaded to the server and the output would be stored in a database on the server. So, the updated kernel would be automatically applied to the server, so they are always using the latest version of it.
- **Developers**: The developer will use the Felucca control center to develop and test the Pharos kernel. The developer would execute the binary executable locally and store the output in a local database. Moreover, the developer has full control access to the Pharos kernel that allows him to dynamically update it and run tests on it, which means that the developer's update on the kernel should be immediately applied to Felucca

2.2 Customer

To maintain the functionality in Pharos, the customer (developer of Pharos) needs to continuously update the Pharos, which requires us to decouple the Pharos kernel to our control layer and an online kernel updating might be a choice.

3 Prioritized Functional Requirements

3.1 Mandatory Requirements

- **Pharos Functionality**: Providing Offline Pharos execution logic, including binary executable file and signature file management, output file storage and download, chain execution of Pharos tools, and concurrency between jobs submitted by users.
- **Complex Execution**: A job can contain several tasks, which are according to executions. All tasks of a job can be executed in order. Besides, pipelined-execution should be supported, which means the output of execution can be the input of the next execution.
- **Termination of Jobs**: User should be able to kill a running job before it finishes.
- **Kernel Update**: Developers may update the Pharos kernel, so the control panel should be independent of the kernel, making it easy to update locally or remotely.
- **History Records Management**: The execution results of all jobs must be stored persistently. Users can browse those data and download the files at any time.

3.2 Nice-to-have Requirements

- **Online Version of Pharos**: Provide an online version for research. Since they don't care much about the security and they are happy to share their data result with others. We need to deploy a central server to handle user requests.
- **Account System**: As there are different types of users, we can use an accounting system to manage their access. For example, analysts can't modify the tools while researchers can upload their customized tools. Furthermore, only developers could modify the Pharos Kernel. On the other hand, using an accounting system can isolate the execution history of different users.
- **Secure Connection**: If the system is deployed on an Internet-access server, everyone can visit the service and the server will become vulnerable. We can use HTTPS to improve the security of communication between users.

3.3 Optional Requirements

- **Customized Tools**: Other than the Pharos Kernel, users(researchers) may want to execute jobs with their own tools. They can upload their own tools and manage them.
- Create a **user-friendly UI** and maintain it with every updating.
- Provide a good **interaction manuscript** for both Pharos tools and Felucca UI.
- **Load balancing optimization** of online version Pharos tools.

4 Non-functional Requirements

- **Isolation:** The binary executable may be dangerous, we need to keep it isolated with user workspace, a better way to do this is to create a container(like docker) for this data.
- **Support Offline Update:** Analysts may use Felucca without access to the Internet, thus they can't update the Pharos Kernel by downloading the latest release. A convenient way to update the Pharos Kernel without the Internet is necessary.
- **Support Local Network Deployment:** As analysts may deploy Felucca on a server with only LAN access, Felucca must support this kind of environment.

5 Technical Constraints

- [TC-1] Binary files should not be stored on the server. (Delete after execution)
- [TC-2] Docker should be installed on the machine where Felucca runs on.
- [TC-3] Results of finished jobs must be tracked and stored in persistent storage.
- [TC-4] Felucca should work without Internet access. (local server)
- [TC-5] Felucca must work with the Pharos Kernel and won't be affected after updates of the Pharos Kernel.

6 Plans of Different Requirements on Storage

- **The stateful one:** A database is required for storing username, password, binaries, and execution records. Users can upload their binaries and then run those binaries with different parameters for multiple times. After users submit a request, they can return soon and check the output from a list of execution records.
- **The stateless one:** A database is not necessary. Each time the user just uploads their binaries with parameters, and then they must wait until the server sends the output back.
- **The mix of above(Our choice):** A database is required but only for storing username, password, and execution records. The server won't save users' binaries thus there is less pressure on storage. Besides, users' binaries may be malicious and thus should not be stored. Users don't need to wait for the output since they can check the record later.

7 Roles of Team Members

Member	Roles
Di Mu	Coordinator, Front-end Developer
Guancheng Li	Notetaker, back-end Developer
Sudi Lyu	Architecture Designer, back-end Developer
Zihao Zhou	Notetaker, back-end Developer

Table 7.1 Roles of Team Members

8 Scope of Work

8.1 Front-end

- Provide UI for users to upload target executable, signatures files, and configure parameters.
- The parameters of pharos tools are complex and with strict limitations. A good UI may help users to set all these parameters in the correct way.
- Allow users to control and execute tools on the server.
- Allow users to observe and download any output of Pharos tools.
- Possible techniques(popular frontend framework): Angular.js Vue.js

8.2 Back-end

- A server that could run all the Pharos tools on the target executable.
- Save the user's target executable, signatures files, and output of tools in a database for persistence.
- Python-based framework(Django) would be better to support the command-line interface.

8.3 Database

- The input format would be JSON and the number of clients is low, a NoSQL light-weighted database(MongoDB) might be better to use.
- Providing data consistency if we have a server cluster.

8.4 Data Format

- RESTful communication between front-end and back-end to separate them for flexibility.
- Pharos tools could output TEXT or JSON under user's demand, and need to convert them to JSON before saving to the database. Besides, JSON files are used by MongoDB to store information and thus it can directly provide the raw documents. Last but not least, Python-based Django can manipulate JSON files conveniently.

8.5 Dataflow

- Users upload executable files, signatures files go through the back-end into the database.
- The user's configure parameters and back-end use Pharos tools and save the output to the database.
- Users look at outputs in the database and download them.

9 Deliverables

The deliverables will be a prototype system to consolidate the pharos analysis tools and any/all supporting artifacts. This will include source code, test cases, design documentation, requirements documentation, and user manuals, and miscellaneous documentation. If the team is successful, then there may also be an opportunity to present their work to the CERT team building the tools.

10 Schedule

Project Milestone	Deliverables	Due Date
Sprint 0	1. Investigate the client's requirements and prioritize them 2. Draw high-level architecture diagrams 3. Design the components and the architecture of the program	Feb 21st, 2020
Sprint 1	1. Finalize the requirements 2. Decide the technologies we are going to use	Mar 13th, 2020
Sprint 2	1. Learning the domain knowledge of pharos tools 2. Improve the use case diagram and requirement 3. Design the skeleton of system architecture	Apr 3rd, 2020
Sprint 3	1. Build a VM to ensure a consistent environment 2. Create a dev-op environment 3. Connecting the technical stack to the requirement 4. Develop a metric for project	Apr 17th, 2020
Sprint 4	1. Break the use case into issues/task 2. Create slides for the final presentation 3. Split the document into requirements and architecture	May 1st, 2020

Table 10.1 Sprint Schedule

11 Planning

11.1 Overview Planning

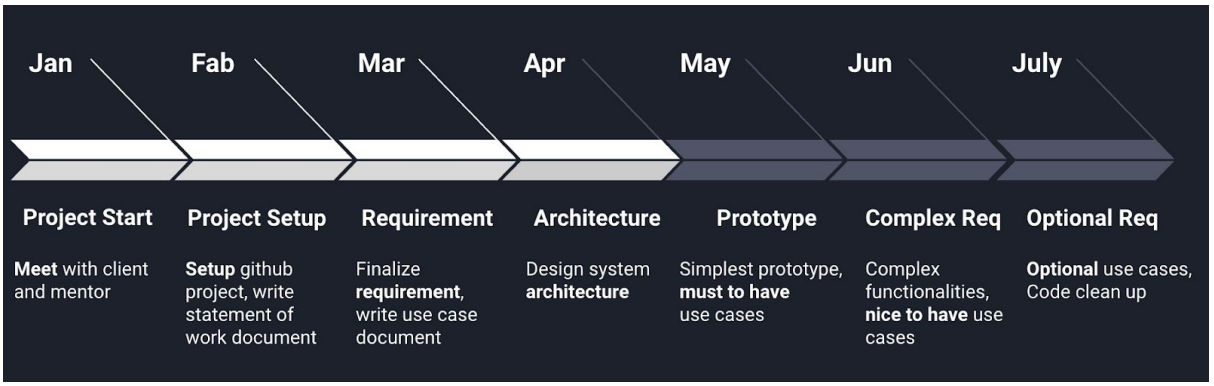

Figure 11.1 Overview Planning

The above graph shows the overview planning for this project, From January to April, we basically start the project, meet with the clients, finalize the requirements, and design the system architecture. From May to July, we need to finish the development work of these projects. In May, we plan to create a prototype for this project, including most of the use cases with must-to-have priority, then we cut a release and demo this to the client. In June, we focus on the complex requirement, finish most of the nice-to-have use cases, then we cut a release and get some feedback from the user. In July, we may focus on some improvement of the system based on the feedback we get before and add some optional requirements.

11.2 Scrum Methodology

Figure 11.2 Scrum Methodology

We use the Scrum Methodology to iterate the project like the graph above.

1. **Client Requirement:** We arrange meetings with the client and finalize the detailed requirements from them, we write or update the requirements document.
2. **Sprint Planning:** We break the requirement into several functional issues, create the issue on GitHub, label them, and plan these issues in the coming sprint.

3. **Development and Testing:** We finish the development work of the issue we created before, create the pull request, do the code review, go through the unit test, integration test, system test, and usability test. Then we merge the code to the master branch.
4. **Release:** We cut the release after we finish these requirements and demo them back to the user, and get the new feedback or requirements.

11.3 Future Sprint

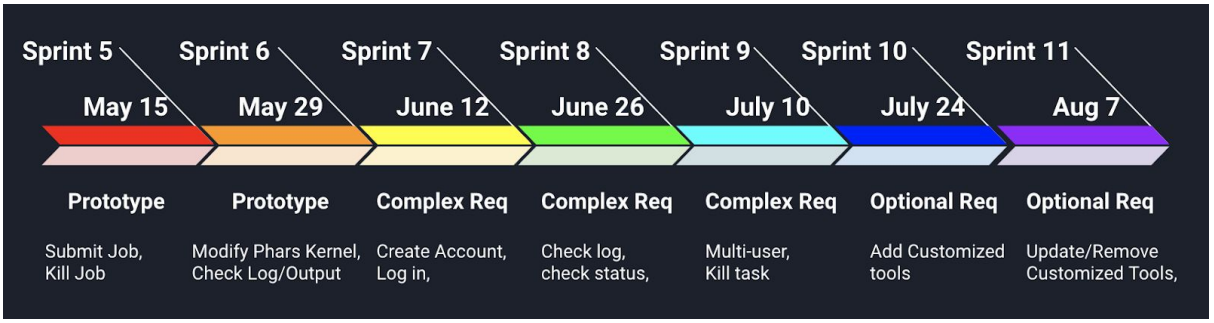

Figure 11.3 Future Sprint

The above figure shows the future sprint plan for the incoming summer semester, we will have 7 sprints in total in the coming three months, The sprint 5 and 6 will mainly focus on prototype development, it needs to finish all the must-to-have use cases. Then we will cut a release, demo to the client, and get feedback. The sprint 7, 8, and 9 should focus on more complex requirements, also we need to add more issues based on the feedback. The sprint 10, 11 will focus on some optional requirements and improve the performance of the project. This sprint plan is a general plan for the future sprint, it can be changed due to some reasons like requirement changes, progress delay, bug fixing, etc.

11.4 Future Issues

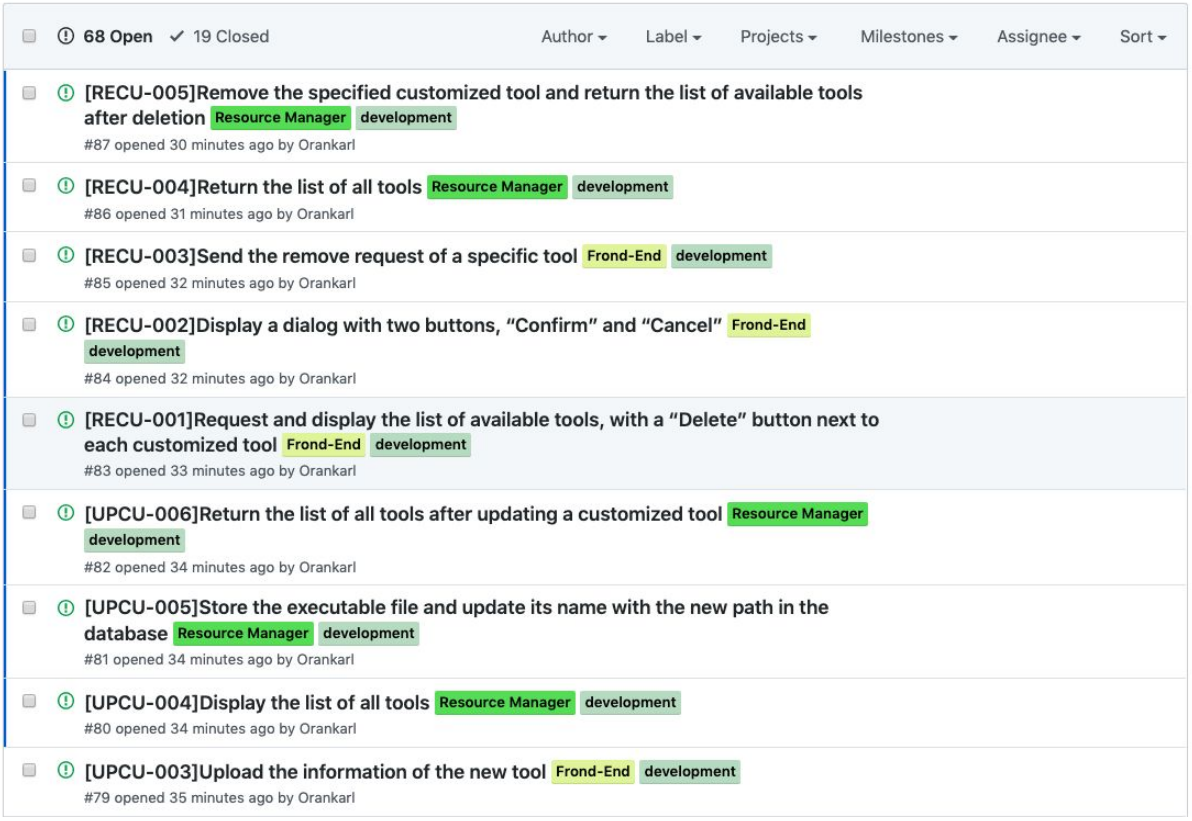

Figure 11.4 Future Issues

Based on the use cases we defined before, we break the use cases into several function issues, create the issue in the GitHub, and carefully label them.
https://github.com/MITS-Felucca/Felucca/issues

12 Tracking

12.1 Team Progress

	Project Setup		Sprint 0		Sprint 1			Spring Break	Sprint 2		Sprint 3		Sprint 4		Total(hours)
			GitHub Setup		Requirments				Documents		Architecture		Presentation		
Date	Jan. 31th	Feb. 7th	Feb. 14th	Feb. 21st	Feb. 28th	Mar. 6th	Mar. 13th	Mar. 20th	Mar. 27th	Apr. 3rd	Apr. 10th	Apr. 17th	Apr. 24th	May. 1st	
SOW document									Finish						28
UseCase doc												Finish			64
Domain Knowledge												Finish			20
Architecture													Finish		70
Tech Stack													Finish		15
Presentation														Finish	32
															229

Figure 12.1 Team Progress

We have the four sprints in this semester from Feb. 14th to May 1st, and we accomplished several documents, the above table shows the entire team progress in this semester.

12.2 Team Metric

	Meetings(25%)	Meeting Notes(%)	Communications(%)	Use cases Doc(%)	SOW Doc(%)	Domain Knowledge(%)	Tech Stack(%)	Architecture Doc(%)	Functional Reqirement(%)	Presentation(%)	SUM
Hour	60	7.5	5	64	28	20	15	70	18	32	319.5
Weight	0.19	0.02	0.02	0.20	0.09	0.06	0.05	0.22	0.06	0.10	1.00
Team member											
Di Mu	25	20	100	25	25	20	25	16.67	25	25	23.91830986
Sudi Lyu	25	0	0	25	25	40	25	33.33	25	25	26.78591549
Zihao Zhou	25	70	0	25	25	20	25	16.67	25	25	23.52707355
Guancheng Li	25	10	0	25	25	20	25	33.33	25	25	25.7687011
Sum	100	100	100	100	100	100	100	100	100	100	100

Figure 12.2 Team Metric

The above figure shows the team metric for each team member. Based on the hours we spend on each project, we define the weight of each task and the percentage of team members who contribute to this task. Then, we calculate the total score for each person to reflect how each team member contributes to this project, and how we balance the workload in our team.

12.3 Kanban Board

We use the Kanban Board to keep track of the issues in GitHub. The below graph shows the snapshot for each sprint. The issues are in three columns, Todo, in progress, and done.

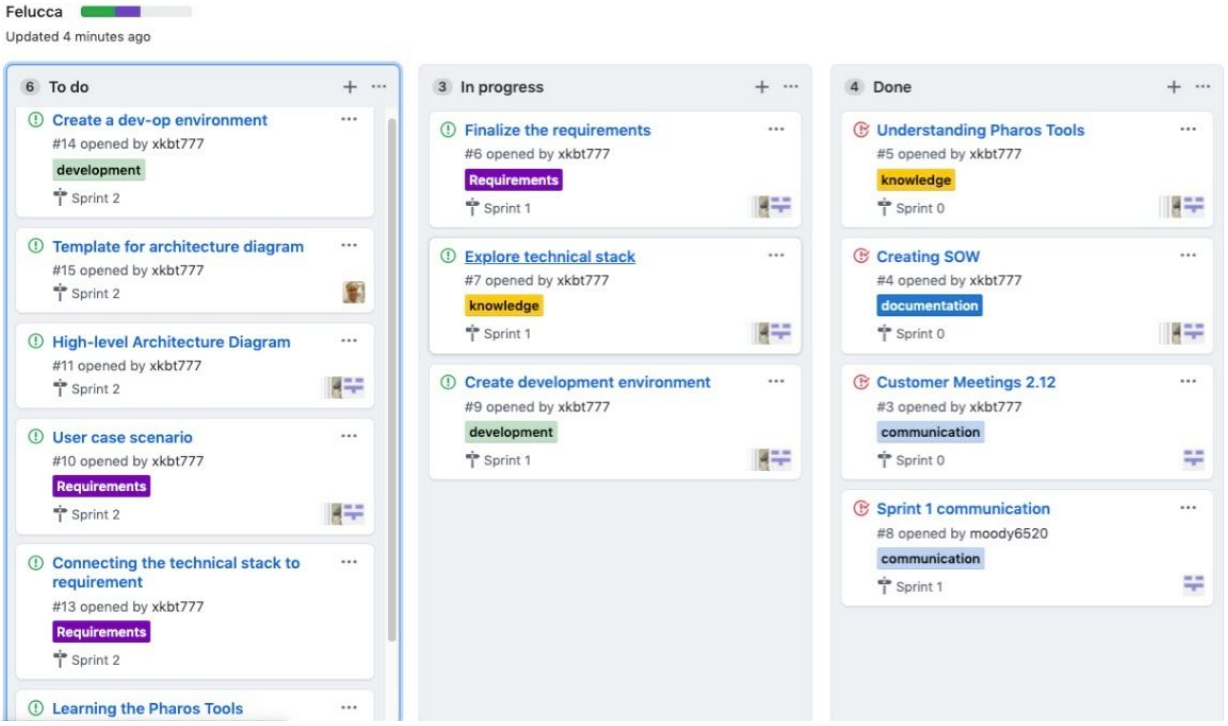

Figure 12.3 Sprint 2 Kanban board

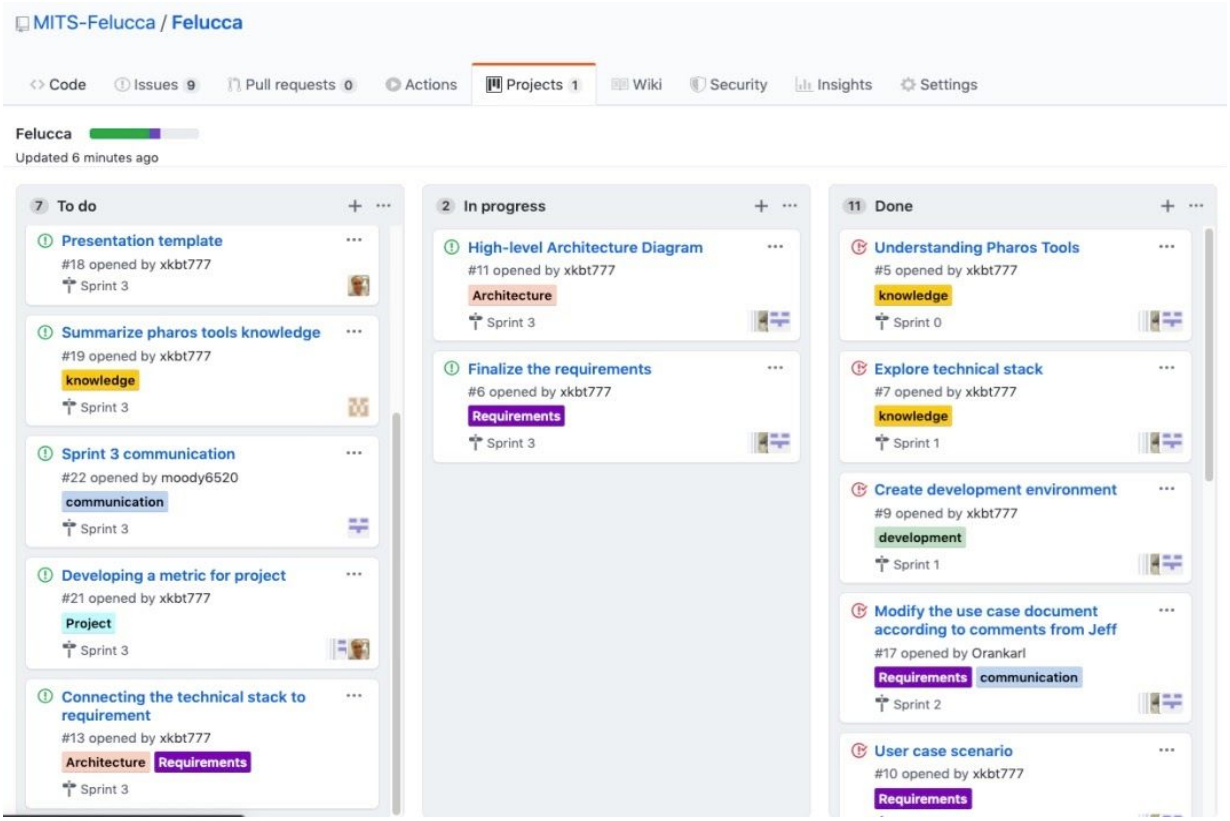

Figure 12.4 Sprint 3 Kanban board

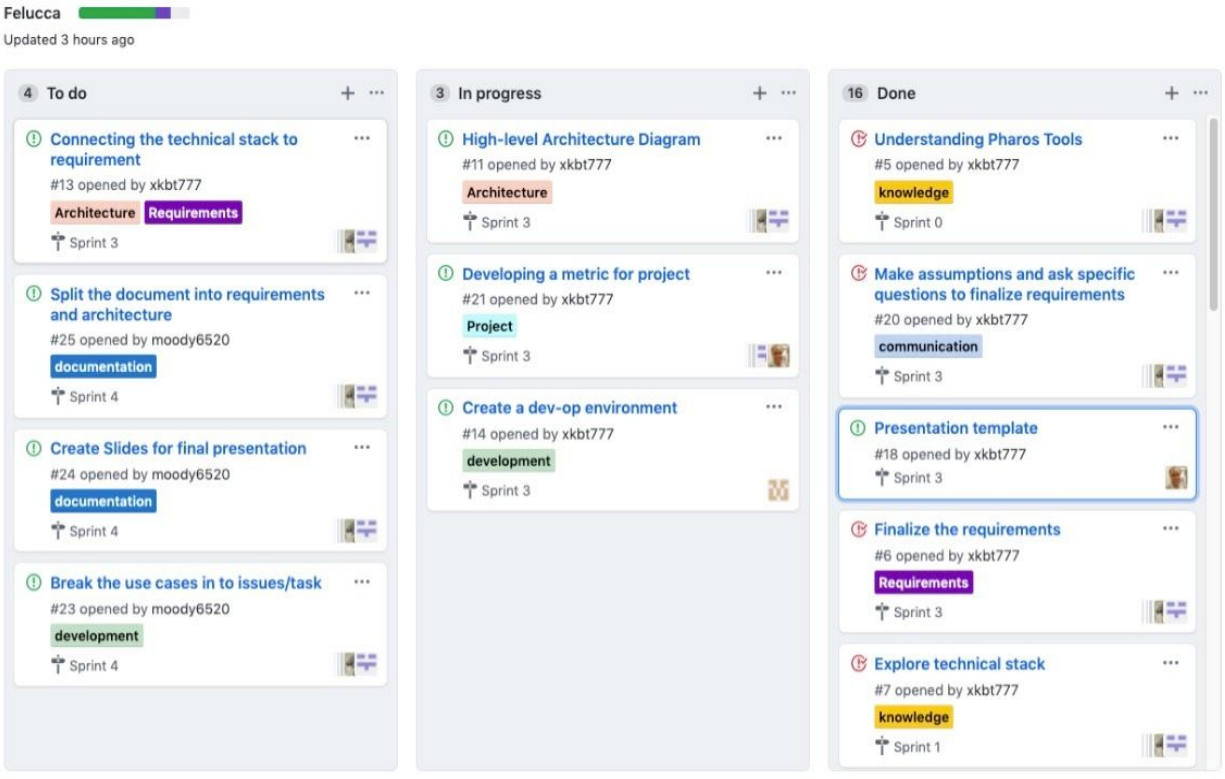

Figure 12.5 Sprint 4 Kanban Board