CST Part IA: Software Engineering and Security, SV 2
Joe Yan
2017-5-18

1 What would influence your decision about splitting the task into modules?
Splitting the task into modules means splitting the whole team into different groups which take responsibilities for different aspects of the project.
In a manpower point of view, a specific group of people just need to learn a relevant part of the job assigned to the group. This reduces the pre-education cost and duration.
Also making a group focus on a specific topic can deepen the group understanding of specific technical issue. This potentially increases the chance of hazard and bug detection which is significant for security engineering.
However, splitting generates interaction costs of communication for agreement between different teams. And human nature makes conflict, blame-passing and bureaucracy between teams which leads to unnecessary energy cost.
Based on the previous assumption, modulus task should be minimum coupling and maximum cohesion. The number and size of each team should be controlled to maximise the efficiency and minimise the negative competition. Also the developer arrangement should be appropriate to smooth the development and avoid specific group having no work to do at some point.
On the testing aspect, a unit testing will be easier for different groups because humans tend to criticise or judge others harsher. And making group taking responsibility for interacting modules test each others' code smooths the later integration and system test because modules communication bugs will be less likely.
Also as it is a security system, the manager can also put more resources to more critical group to minimise the potential safety issue.

2 Would you use waterfall, spiral or something else to implement FALCON? Waterfall model spilts the development into requirement, specification, implementation and unit test, integration and system test and operation and maintenance. This model progressively refines the requirement. The critical factor is that all the requirement should be fully decided at the beginning of the project.
Spiral model take several iteration for the development and solve a group of requirement or specification in one iteration. This brings a nice factor that the requirement is not necessarily fully defined before the development begins. The user can add new requirements later. Also there is a better risk management due to multiple development iterations.
Evolution model makes the further development on the project possible. And opening the source can potentially attract more developer to add new features to the project or debug which makes the project self-evolve under the users' requirement.
The implementation of FALCON is actually an evolution of an existed module of a project so evolution model does not make sense for a small module being evolved. Also as an authorisation modulus, all the requirement is clear at the very beginning so the waterfall model is a reasonable choice which works well with many development and management tools.

3 Would you deploy it using versioned software or continuous deployment? The version software is a sensible choice here. Because the Kudos is a released system when new development going on, the old version will still be on line. Keeping the old version as a main stem and making a new branch for the current development gives the space for the current development and also keep the old version which ensure that the old version is

still capable for debugging and emergency fixing. For example if a bug makes supervisor unable to post new work and the version control is not implemented it will be a mess to fix the bug due to the lost of track of the old version.

4 What modern tools would help to reduce development bugs? (Explain how each would help – what it helps to avoid and what it costs you to use it.)
A version control tool is needed as discussed before it also shares the code in the team.
An IDE can speed up the coding by inner documentation and auto-fill.
A compilation and linking tool can make the assemble of different parts of the project easier.
A documentation generator can reduce the effort of writing the document.
A data stream visualiser (possibly?) can clarify the communication channel and increase the chance of hazard detection.
A debugger with bug database can quickly check all common found bugs and spot the same type of bug in different place of codes.
A memory debugger can detect potential stack overflow and out-range memory access.
A unit tester can tests suspicious inputs and report which may bring hazards.
The efficiency and cost ratio is important here. The cost includes buying tools and educating the developer to use tools. Picking tools wisely is critical. An over-powered tool just brings negative effect to the development. For example, a tool with high price and take each developer a long period to fully handle it but just bringing a small risk reduction and efficiency. Then it is probably not worth being used.

5 How would you structure/design/implement the project now to make future maintenance (debugging, bug fixing, enhancements, etc.) as bug-free as possible?
The source code should be highly readable which includes aspects of names of variable, indentation and annotation. A code review session is helpful.
The project should be well documented making the later maintenance have a reliable reference.
Libraries used should be as popular, well-supported and well-documented as possible which share others efforts to detect and fix the potential bug.
All debugging information during the development should be kept because the same or similar bug may appear again or simply missed during the development debugging.
The server should be reliable. A terrible server may be caused by a mindless server maintainer or bad physical supports (e.g. frequently power down or low connection quality).
Data backup is important. If the data loss does happen, we can still roll back the database to minimise the loss.

6 How might predicting your users' behaviour make your product more secure in practice?
Users of Kudos are intelligent and well-educated. This makes them either sensible to the security issue or "intelligently" setting a weak password because of their confident cost and risk judgement (creating and remembering a long password comparing with unlikely interests of supervision booking information). Without research support, it is hard to classify the mental model of this rather small and specific group.
As said above, the password is the main issue of an authorisation system.
If the system does not restrict a strong password for users weak passwords will be abused.
If the system does (e.g. generate a long random password for them) users will probably just write them done which is unsafe or blaming the designer (although they do not have a second choice here).
One possible approach is to use cheese model to defend in depth to minimise the hazard of a weak password. For example, once the login request does not come from the University of Cambridge IP or not from the usual IP address, machine, OS or browser. The system can ask the user what is their next closest supervision to further the authorisation. Sending a email to the assigned email with a verification code is another approach.