Suggestions for adding defensive programming tactics to the 345 class project

This semester’s project description is the following:

“Faculty members often notify their students via e-mail regarding their final grades. The e-mail usually contains information other than just the final grade. For example, it may show the student’s grades that have been recorded throughout the semester. The faculty member may even make some comments regarding the student’s performance.

Your job is to automate this process.”

In an effort to encourage defensive programming tactics to the students we thought that part of the System Requirements Specification (SRS) could include possible security issues related to the student’s project. Typically these types of concerns are addressed with shall-not type requirements. For instance:

“The automated grade reporter shall not contain any buffer overflows.”

Or

“The automated grade reporter shall not place an overwhelming burden on the university email infrastructure.”

Etc.

Mrs. Mitchell’s comments: While these types of requirements may be a little inappropriate for the higher level SRS, they could be included in the general direction that the programmers might need.

Both this project, and last year’s day-planner project, involve sensitive personal data. Including a requirement for some sort of encryption into the project and how that would be accomplished would be a good exercise. The students wouldn’t have to implement their own crypt-algorithm but could find a decent implementation on the web and incorporate it into their SRS and Software Design Document (SDD). Not only would such a requirement teach them about integrating cryptography into their project, but it would also expose them to using cryptography for their own personal use. Additionally it will give them experience in searching for pre-developed components available on the web.

Mrs. Mitchell’s comments: This type of a requirement may be out of scope for the 345 students, but would provide a great learning experience for 445 students in the future.

In light of the time requirements and the vastness of the semester programming project in general it might not be feasible to test for these defensive programming requirements. But the fact that the students considered such problems and offered tenable solutions, demonstrates an awareness of defensive programming issues and prepares them to fulfill such requirements in the real world.

During the testing and code inspection process students should be required/recommended to run some sort of static analysis tool against their code. Examples are ITS4, a command
line tool that can be integrated into emacs or run in a stand-alone mode, and /GS a tool that has been integrated into MS Visual C++ .Net 2003. Either tool will parse the students’ code into tokens and inspect it for possible development flaws that could lead to security vulnerabilities.

**Mrs. Mitchell’s comments:** Levying a requirement like this might be more then the students of 345 could handle and would likely be pushed to 445.

Within the Code Inspection Report (CIR), a section on defects has already been included. This is a great launching point for pointing out how those defects discovered in development could lead to security concerns down the road. Frequently students discover a bug in their code that causes it to crash. They fix the bug if it is a huge problem or ignore it if it only arises due to a peculiar set of circumstances. Impressing upon students the danger of such bugs is a key aspect of teaching defensive programming tactics. Asking the students to consider how any of their defects could lead to a security vulnerability would require some critical thinking by the students but would not overburden them while trying to complete the project.