Top Ten Best Practices for Secure Coding

To ensure that you build and deploy the most secure software possible, Risk Factory recommends the following best practices for secure coding:

1. **Validate user input**

   Never assume that input from HTML forms is valid. Just because you gave the user only hidden fields or pull downs, or you had JavaScript to validate every input, does not guarantee the input will not be tampered with. A hacker with a simple local web proxy can change anything they want after the JavaScript executes.

   Verify that all input from the user contains valid characters and represents a valid value before using that value in your application. Check it early in the processing of the request to avoid accidental use before the validation is done.

   Also, check it at multiple levels (see defence-in-depth, later). Be restrictive. You can always ease up on the restrictions. It's harder to tighten up the validation rules later, as you may have already stored user data that would no longer be valid.

2. **Escape input values**

   Implementing any code that creates file paths, HTML, SQL statements or other strings that another subsystem parse requires care. User input may contain characters that allow a hacker to cause your application to pass invalid strings to those subsystems that result in unauthorised access.

   Unless you disallow all special characters in your input validation, you will need to make sure you properly escape or in some other way, account for special characters in the target subsystem (see the documentation for the subsystem in question for definitive escape requirements). For example,

   - in inputs used to calculate file paths: /, \. or ..
   - in strings used to calculate or display HTML: >, <, ", &
   - in strings used to calculate SQL statements: ', ", \

   Many runtime environments already provide functions to escape these inputs. Use techniques like parameter substitutions in the database interface rather than building up query strings using concatenation. Leverage this and other layers of protection that system components provide.

3. **Fail Safe**

When making decisions that affect security, it is important to write code to deny access by default. Only allow access after confirmation that the user has proper authorisation to proceed. Also, avoid the use of negatives. It is too easy to get the sense of logic wrong. The code above would be clearer if the local variable reflected the same sense as that of the method being implemented.

4. **Treat sensitive security information with care**

   Be mindful of the type of information being handled. If the information is sensitive, take special care in the code that handles it. Breaches that reveal information such as passwords, PINs and personal data can be disastrous.

   Make sure this information is only stored in appropriate locations. For instance, never write to an application or system log any of the user's personal information (password, SSN, credit card numbers, etc.), as these logs may be readable by operational personnel who should not have access to that personal information. Write only enough information about the user to identify within the application which user made the request.

   Encrypt the sensitive data stored on mass storage. Storing clear-text passwords in a database, for instance, means that a hacker who simply gains read access may have all the keys to the kingdom.

   Only store the hashed copy of the password and use this to compare with the hashed user input (these values should also be salted to increase the work required for a dictionary attack).

   Inevitably, there are clear-text copies of the user input in memory, often on the heap long after the code runs, and maybe on the paging file of the operating system. Keep the length of the code path used to process the clear-text password as short as possible. Also, clear the contents of this local memory storage within the same block as the declaration of that storage. This will help keep the clear-text passwords off the stack and heap.

5. **Practice “Defence-in-Depth”**

   Protect your application in multiple ways. Do input validation with the tools the application environment provides. In addition, write your code to assume the input validation might fail. This costs little in terms of code or performance but makes the code more robust in the face of failure.

6. **Minimise error message information**

Provide intelligible, useful error messages to your users, but keep the details in the log file. For example, users don't need to know about database operations or details of the errors encountered. A safer approach is to tell the user something went wrong, but only provide the details in the log.

7. **Good comments**

   Good comments in code are a major help in maintaining code – if they help make the code clearer. Many developers know they should add comments and so they take the easy way out and add comments saying what the code does. The code is right there. Readers can see what it does – these comments are not what they need. Tell them why decisions are being made the way they are. Help the reader understand the code.

8. **Study patterns**

   When reviewing code, you can often find logic errors that may affect security by watching for patterns in the source code and looking for exceptions to those patterns. For instance, in the following code:

```
if (result == CASE_1) return(VALUE_1);
else if (result == CASE_2) return(VALUE_2);
else if (result == CASE_3) return(VALUE_3);
else if (result == CASE_4) return(VALUE_4);
else if (result == CASE_5) return(VALUE_2);
/* otherwise we're good to go with the default */
return (VALUE_0);
```

   The next to last return statement looks suspicious. It could be a cut and paste error. It could be a logic error and cause the return value to mislead the caller into providing access to something that should be denied – e.g., a list of employees rather than a list of distributors.

9. **Have someone review your code**

   Find someone else to review your code and offer to reciprocate. Having someone else read through your code almost always results in them asking you questions. The 'what if?' and 'why that way?' questions make you think about and justify your choices outside the more solitary activity of writing the code. This change in context provides an opportunity to step back and take a fresh look at your work.

10. **Study defects**

    When a security-related defect is found in your code, try to understand the cause. Study the defect and try to determine how you would have done things differently. When another developer makes the change,

understand the changes that they made – maybe there's another way to fix it other than what you chose.

Over time you'll be able to recognise the patterns of these defects and generalise the solutions. Part of what you should strive for is to not make the same mistake.

Learn from your mistakes and the mistakes of others.