Security Throughout the Software Development Process

Generally speaking, we should think of security is an ongoing process. For best results, it helps to integrate security into all phases of the system development lifecycle: requirements analysis, design, implementation, testing, quality assurance, bug-fixing, and maintenance. It's particularly crucial to appreciate that security is not a feature or checklist item that can be effectively bolted-on after the software has been developed.

Test code thoroughly before deployment.

Testing can help eliminate bugs. It is worth putting effort into developing test cases that might trigger security bugs or expose inadequate robustness. Test corner cases: unusually long inputs, strings containing unusual 8-bit characters, strings containing format specifiers (e.g., `%s`) and newlines, and other unexpected values. Manuals and documentation can provide a helpful source of potential test cases. If the manual says that the input must or should be of a particular form, try constructing test cases that are *not* of that form.

Unit tests are particularly valuable at checking whether you are doing a good job of defensive programming. Try inputs that stress boundary conditions (for integers, for example, 0, 1, -1, $2^{31}-1$, and -2^{31} are fun to try). If the routine operates on pointers, try inputs with unusual pointer aliasing or pointing to overlapping memory regions.

Automate your tests, so that they can be applied at the push of a button. Run them nightly.

Use code reviews to cross-check each other.

Good security programmers enlist others to review their code, because they realize that they are fallible. Having someone else review your code is usually much more effective than reviewing your own code. Bringing in another perspective often helps to find defects that the original programmer would never found. For instance, it is easy to make implicit assumptions (e.g., about the threat model) without realizing it. The original programmer is likely to make the same erroneous assumption when reviewing their own code as when they wrote it, while someone else may spot the flaw immediately. Knowing that someone else will review your code also helps keep you honest and motivates you to avoid dangerous shortcuts, because most people prefer not to be embarassed in front of their peers.

Evaluate the cause of all bugs found.

What should you do when you find a security bug? Fix it, obviously—but don't stop there.

First, generate a regression test that triggers the security hole. Add it to your regression test suite so that if the bug is ever re-introduced you will discover it very quickly.

Second, check whether there are other bugs of a similar form elsewhere in the codebase. If you find three or four bugs of the same type, it is good bet that there are more lurking, waiting to be found. Document the pitfall or coding pattern that causes this bug, so that other developers can learn from it.

Third, evaluate what you could be doing differently to prevent similar bugs from being introduced in the future. Does the bug reveal a misfeature in your API? If so, fix the API to prevent any further incidence of such bugs.

You may also wish to investigate the root cause of such bugs periodically. Are there adequate resources for security? Is security adequately prioritized? Was the design well-chosen? Are you using the right tools for the job? Are deadlines too tight and programmers feeling too rushed to put adequate care into security concerns? Does it indicate some weakness in the process you use? Do engineers need more training on security? Should you be doing more testing, more code reviews, something else? Even if you fix each security bug as they occur, if you don't fix the root cause that creates the conditions for such bugs to be introduced, then you will continue to suffer from security bugs.

Use "defensive programming".

The goal of *defensive programming* is to ensure that your module will remain robust even if all other modules that interact with it misbehave. The general strategy is to assume that an attacker is in control of the inputs to your module, and make sure that nothing terrible happens.

The simplest situation is where we are writing a module M that provides functionality to a single client. Then M should strive to provide useful responses as long as the client provides valid inputs. If the client provides an invalid input, then M is no longer under any obligation to provide useful output; however, M must still protect itself (and the rest of the system) from being subverted by malicious inputs.

A very simple example:

```
char charAt(char *str, int index) {
    return str[index];
}
```

This function is fragile. First, `charAt(NULL, *any*)` will cause the program to crash. Second, `charAt(s, i)` can create a buffer overrun situation if `i` is out-of-bounds (too small or too large) for the string. Neither can be easily fixed without changing the function interface.

Another made-up example:

```
char *double(char *str) {
    size_t len = strlen(str);
    char *p = malloc(2*len+1);
    strcpy(p, str);
    strcpy(p+len, str);
    return p;
}
```

This function could potentially be criticized on several grounds:

- `double(NULL)` will cause a crash. Fix: test whether `str` is a null pointer, and if so, return null.
- The return value of `malloc()` is not checked. In an out-of-memory situation, `malloc()` will return a null pointer and the call to `strcpy()` will cause the program to crash. Fix: test the return value of `malloc()`.
- If `str` is very long, then the expression `2*len+1` will overflow, potentially causing a buffer overrun. For instance, if the input string is 2^{31} bytes long, then on a 32-bit machine we will allocate only 1 byte, and the `strcpy` will immediately trigger a heap overrun.

A trickier example: Consider a Java sort routine, which accepts an array of objects that implement the interface `Comparable` and sorts them. This means that each such object has to implement the method `compareTo()`, and `x.compareTo(y)` must return a negative, zero, or positive integer, according to whether `x` is less, equal, or greater than `y` in their class's natural ordering (e.g., strings might use lexicographic ordering, say). Implementing a defensive sort routine is actually fairly tricky, because a malicious client might supply objects whose `compareTo()` method behaves unexpectedly. For instance, calling `x.compareTo(y)` twice might yield two different results (if `x` or `y` are malicious or misbehaving). Or, we might have `x.compareTo(y) == 1`, `y.compareTo(z) == 1`, and `z.compareTo(x) == 1`, which is nonsensical. If we're not careful, the sort routine could easily go into an infinite loop or worse.

Here is some general advice:

- *Check for error conditions.* Always check the return values of all calls (assuming this is how they indicate errors). In languages with exceptions, think carefully about whether the exception should be handled locally or should be propagated and exposed to the caller. Check error paths very carefully: error paths are often poorly tested, so they often contain memory leaks and other bugs.

  What do you do if you detect an error condition? Generally speaking, for errors that are expected and intended to be recoverable, you may wish to recover. However, unexpected errors are by their very nature more difficult to recover from. In many applications, it is always safe to abort processing and terminate abruptly if an error condition is signalled; *fail-stop* behavior may be easier to get right.

- *Don't crash or enter infinite loops. Don't corrupt memory.* Generally, you will want to verify that, no matter what input you receive (no matter how peculiar), the program will not terminate abnormally, enter an infinite loop, corrupt its internal state, or allow its flow of control to be hijacked by an attacker. Be sure that these failures cannot happen. Trust no one. If there are any inputs to this function, validate its inputs explicitly to avoid these cases (even if you are not aware of any caller that could provide such bad inputs).

  If availability is important, you will generally need to avoid leaking memory or other resources, since once enough memory is leaked, the program might cease to operate usefully. You may also want to defend against algorithmic denial-of-service attacks: if the attacker can supply inputs that lead to worst-case performance that is far worse than the normal case, this can be dangerous. For instance, if your program uses a hash table with $O(1)$ expected time per lookup, but $O(n)$ worst-case time, the attacker might provide inputs to it that trigger the $O(n)$ worst-case behavior and cause the program to essentially freeze as it enters a protracted computation.

- *Beware of integer overflow.* Integer overflow often violates the programmer's mental model and leads to unexpected—and hence often undesired—behavior. You should consider verifying that integer overflow is impossible.

- *Check exception-safety of the code.* In languages with exceptions, there are usually two kinds of exceptions: those explicitly thrown by a programmer, and those implicitly thrown by the platform if some runtime error is detected. For instance, a null pointer dereference, a division by zero, an invalid cast, or an out-of-bounds array reference each trigger a runtime exception. Generally, you should verify that your code will not throw a runtime exception under any circumstance, because such exceptions are usually indications of unexpected behavior or program bugs. Less restrictively, one might check that all such exceptions are handled and will appropriately propagate across module boundaries.

  A famous example of a failure to verify exception-safety comes from the Ariane rocket. The Ariane 4 contained flight control software written in Ada. When the more powerful version, Ariane 5, was developed, the same software was reused. Unfortunately, when the Ariane 5 was launched, it blew up shortly after launch. The cause was discovered to be an uncaught integer overflow exception, which caused the software to terminate abruptly. A certain 16-bit register held the horizontal velocity of the flight trajectory. On the Ariane 4, it had been verified that the range of physically possible flight trajectories could not overflow this variable, so there was no need to install an exception handler to catch such an exception. However, the Ariane 5's rocket engine was more powerful, causing a larger horizontal velocity to be stored into the register and triggering an overflow exception that crashed the on-board computers. The assumption made during the construction of the Ariane 4 was never re-validated when the software was re-used in the Ariane 5, causing losses of around $500 million.

How does defensive programming relate to the use of *preconditions*?[1] Of course, whenever we want to make some assumption about the calling context, we can either express this as a precondition and leave it to the caller to ensure it is true, or we can explicitly check for ourselves that the condition holds (and abort if it does not). How should we decide between these two strategies? Perhaps the most sensible approach is to use preconditions to express constraints that honest clients are expected to follow. So long as the client meets the documented preconditions (whether formal or informal), then the module is obligated to return correct and useful results to the client. If the client departs from the documented contract, then the module is no longer under any obligation to return useful results to that client, but it still must protect itself and other clients. Thus, for interfaces exposed to clients, we might (a) use documented preconditions to express the intended contract and (b) use explicit checking for anything that could corrupt our internal state, cause us to crash, or disrupt other clients. For internal helper functions that can only be invoked by code in the same module, we might not worry about the threat of being invoked with malicious inputs, and we could freely choose between implicit checking (preconditions) and explicit checking.

[1] See the *Reasoning About Code* notes for a discussion of preconditions.