CS 161 Computer Security
Fall 2005 Joseph/Tygar/Vazirani/Wagner **Notes 15**

Writing Secure Code

This lecture discusses implementation techniques to avoid security holes when you write code. We will describe many good practices. Many of these have a strong overlap with software engineering and general software quality, but the demands of security place a heavier burden on programmers.

In security applications, we must eliminate *all* security-relevant bugs, no matter how unlikely they are to be triggered in normal execution, because we are facing an intelligent adversary who will gladly interact with our code in abnormal ways if there is any profit in doing so. Compare to software reliability, where we normally focus on the bugs that are most likely to happen; bugs that only come up under obscure conditions might be ignored if reliability is the goal, but they cannot be ignored when security is the goal. Dealing with malice is much harder than dealing with mischance.

In these notes, we'll especially emphasize three fundamental techniques: (1) modularity and decomposition for security; (2) formal reasoning about code using invariants; (3) defensive programming. At the end, we also discuss programming language-specific issues and integrating security into the software lifecycle.

1 Modularity

A well-designed system will be decomposed into modules, where modules interact with each other only through well-defined interfaces. Each module should perform a clear function; the essence is conceptual clarity of what it does (what functionality it provides), not how it does it (how it is implemented).

The granularity of modules is dependent on the system and language. A module typically has state and code. For instance, in an object-oriented language like Java, a module might consist of a class (or a few closely related classes). In C, a module might be in its own file and contain some clear external interface, along with many internal functions that are not externally visible or callable.

Module design is as much about interface design as anything else. The interface is the contract between caller and callee; hopefully, it should change less often than the implementation of the module itself. A caller should only need to understand the interface. Modules should interact only through the defined interface; for instance, you shouldn't use global variables to communicate information from caller to callee. Think of a module as a blob; the interface is its surface area, and the implementation is its volume. Thoughtful design is often characterized by narrow and conceptually clean interfaces and modules with a low surface area to volume ratio.

When you decompose the system into modules, here are some suggestions that will improve security:

- *Minimize the harm that could be caused by failure of a module.* Ensure that even if one module is penetrated (e.g., by a buffer overrun) or behaves unexpectedly (e.g., due to a bug in its implementation), then the damage is contained as much possible. Draw a security perimeter around each module. Protect modules from each other, so that one misbehaving module cannot cause other modules's behavior to deviate from what was expected by the programmer. Plan for failure: think in advance about what the consequences of a compromise of each module might be, and structure the system to reduce these consequences.

For instance, a monolithic architecture that places all modules in a common address space is an unnecessary security risk, because if one module is compromised then all others can be penetrated as well. Some languages (e.g., Java) provide mechanisms for isolating modules from each other using type-safety; with legacy languages (like C), you may need to place each module in its own process to protect it.

- *Follow the principle of least privilege at a module granularity.* Provide each module the least privilege that is necessary to get its job done. Architect the system so that most modules need only minimal privileges.

  Think about whether there is a way to structure the system so that the complex computations that will require a lot of code are isolated in modules with few privileges. Modules with extra privileges should have very little code. The more privilege a module is given, the greater the confidence we will want to have that it is correct, and more confidence generally requires less code.

Example: A network server that listens on a port below 1024 might be broken up into two pieces: a small start-up wrapper, and the application itself. Because binding to a port in the range 0–1023 requires root privileges, the wrapper could run as root, bind to the desired port to some file descriptor, and then spawn the application and pass it the file descriptor. The application itself could then run as a non-root user, limiting the damage if the application is compromised. The wrapper can be written in only a few dozen lines of code, so we should be able to validate it quite thoroughly.

Example: A web server might be structured as a composition of two modules. One module might be responsible for interacting with the network; it could handle incoming network connections and parse them to identify the requested URL. The second module might translate the URL into a filename and read it from the filesystem. Note that the first module can be run with no privileges at all (assuming it is started by a root wrapper that binds to port 80). The second module might be run as some special userid (e.g., `www`), and we might ensure that only documents intended to be publicly visible are readable by user `www`. This then leverages the file access controls provided by the operating system so that even if the second module is penetrated, the attacker cannot cause any harm to the rest of the system.

2 Reasoning About Code

Often functions make certain assumptions about their arguments, and it is the caller's responsibility to make sure those assumptions are valid. These are often called *preconditions*. A precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`. The function `f()` is supposed to behave correctly and produce meaningful output as long as its preconditions are met. If any precondition is not met, all bets are off. Therefore, the caller must be sure to call `f()` in a way that will make these preconditions true. In short, a precondition imposes an obligation on the caller, and the callee may freely assume that the obligation has been met.

Here is a simple example of a function with a precondition:

```
/* Requires: p != NULL */
int deref(int *p) {
    return *p;
}
```

It is not safe to dereference a null pointer; therefore, we impose a precondition that must be met by the caller of `deref()`. The precondition is that $\texttt{p} \neq \texttt{NULL}$ must hold at the entrance to `deref()`. As long as all callers ensure this precondition, it will be safe to call `deref()`.

Assertions may be combined using logical connectives (and, or, implication). It is often also useful to allow existentially ($\exists$) and universally ($\forall$) quantified logical formulas. For instance:

```
/* Requires:
    a != NULL
    for all j in 0..n-1,  a[j] != NULL */
int sum(int *a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

The second precondition might be expressed in mathematical notation as something like

$$\forall \texttt{j} \,.\, (0 \leq \texttt{j} < \texttt{n}) \implies \texttt{a[j]} \neq \texttt{NULL}.$$

If you are comfortable with formal logic, you can write your assertions down in this way, and this will help you be precise. However, it is not necessary to be so formal. The primary purpose of preconditions is to help you think explicitly about precisely what assumptions you are making, and to communicate those requirements to other programmers and to yourself.

Postconditions are also useful. A postcondition for `f()` is an assertion that is claimed to hold when `f()` returns. The function `f()` has the obligation of ensuring that this condition is true when it returns. Meanwhile, the caller may freely assume that the postcondition has been established by `f()`. For example:

```
/* Ensures: retval != NULL */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) {
        perror("Out of memory");
        exit(1);
    }
    return p;
}
```

When you are writing code for a function, you should first write down its preconditions and postconditions. This specifies what obligations the caller has and what the caller is entitled to rely upon. Then, verify that, no matter how the function is called, as long as the precondition is met at entrance to the function, then the postcondition will be guaranteed to hold upon return from the function. You should prove that this is always true, for all inputs, no matter what the caller does. If you can find even one case where the caller provides some inputs that meet the precondition, but the postcondition is not met, then you have found a bug in either the specification (the preconditions or postconditions) or the implementation (the code of the function you just wrote), and you'd better fix whichever is wrong.

How do we prove that the precondition implies the postcondition? The basic idea is to try to write down a precondition and postcondition for every line of code, and then do the very same sort of reasoning. The requirement is that each statement's postcondition must match (or imply) the precondition of any statement that follows it. Thus, at every point between two statements, you write down an *invariant* that should be true any time execution reaches that point. The invariant is a postcondition for the preceeding statement, and a precondition for the next statement.

It is pretty straightforward to tell whether a statement in isolation fits its pre- and post-conditions. For instance, a valid postcondition for the statement "`v=0;`" would be $\texttt{v} = 0$ (no matter what the precondition is). Or, if the precondition for the statement "`v=v+1;`" is $\texttt{v} \geq 5$, then a valid postcondition would be $\texttt{v} \geq 6$. As another example, if the precondition for the statement "`v=v+1;`" is $\texttt{w} \leq 100$, then $\texttt{w} \leq 100$ is also a valid postcondition (assuming `v` and `w` do not alias).

This leads to a very useful concept, that of *loop invariants*. A loop invariant is an assertion that is true at the entrance to the loop, on any path through the code. The loop invariant has to be true before every iteration to the loop. To verify that a condition really is a valid loop invariant for the loop, you treat the condition as both a pre-condition and a post-condition for the loop body.

Let's try an example. Here is some code that computes the factorial function:

```
/* Requires: n >= 1 */
int fact(int n) {
    int i, t;
    i = 1;
    t = 1;
    while (i <= n) {
        t *= i;
        i++;
    }
    return t;
}
```

A prerequisite is that that input must be at least 1 for this implementation is not correct. Suppose we want to prove that the value of `fact(.)` is always positive. We'll annotate the code with invariants:

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int i, t;
            /* n>=1 */
    i = 1;
            /* n>=1 && i==1 */
    t = 1;
            /* n>=1 && i==1 && t==1 */
    while (i <= n) {
            /* 1<=i && i<=n && t>=1   <-- loop invariant */
        t *= i;
            /* 1<=i && i<=n && t>=1 */
        i++;
            /* 2<=i && i<=n+1 && t>=1 */
    }
            /* i>n && t>=1 */
```

```
    return t;
}
```

How do we verify that the invariants are correct? This might look pretty complicated, but don't get discouraged—it's actually pretty easy if you just take the time to look at each step. Notice that the function's precondition implies the invariant at the beginning of the function body. Also, the invariant at the end of the function body implies the function's postcondition. Thus, if each statement matches the invariant immediately before and after it, everything will be ok. The only non-trivial reasoning is in the loop invariant. First, we must prove that at the entrance to the first iteration of the loop, the loop iteration will be true, and this follows since the logical proposition $\mathtt{n} \geq 1 \wedge \mathtt{i} = 1 \wedge \mathtt{t} = 1$ implies $1 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{t} \geq 1$ (e.g., if $\mathtt{i} = 1$, then certainly $\mathtt{i} \geq 1$). Also, we must prove that if the loop invariant holds at the beginning of any iteration of the loop, then it will hold at the beginning of the next iteration, if there is another iteration. This is true, since the invariant at the end of the loop body ($2 \leq \mathtt{i} \leq \mathtt{n}+1 \wedge \mathtt{t} \geq 1$) together with the loop termination condition ($\mathtt{i} \leq \mathtt{n}$) implies the invariant at the beginning of the loop body ($1 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{t} \geq 1$). It follows by induction on the number of iterations that the loop invariant is always true on entrance to loop body. The conclusion is that `fact()` will always make the postcondition true, so long as the precondition is established by its caller.

To give you some more practice, we'll show another example implementation of `fact()`, this time using recursion. Here goes:

```
/* Requires: n >= 1 */
int fact(int n) {
    int t;
    if (n == 1)
        return 1;
    t = fact(n-1);
    t *= n;
    return t;
}
```

Do you see how to prove that this code always outputs a positive integer? Let's do it:

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int t;
    if (n == 1)
        return 1;
            /* n>=2 */
    t = fact(n-1);
            /* t>=0 */
    t *= n;
            /* t>=0 */
    return t;
}
```

Before the recursive call to `fact()`, we know that $\mathtt{n} \geq 1$ (by the precondition), that $\mathtt{n} \neq 1$ (since the if statement didn't follow its then branch), and that $\mathtt{n}$ is an integer. It follows that $\mathtt{n} \geq 2$, or that $\mathtt{n}-1 \geq 1$.

That's very good, because it means the precondition is met when making the recursive call, and thus we're entitled to conclude that the return value from `fact(n-1)` is positive (by virtue of the postcondition for `fact()`). The rest is straightforward.

In general, any time we see a function call, we have to verify that its precondition will be met. Then we are entitled to conclude that its postcondition holds, and to use this fact in our reasoning.

If we annotate every function in the program with pre- and post-conditions, this allows *modular reasoning*. This means that I can verify function `f()` by looking only at the code of `f()` and the annotations on every function that `f()` calls—but I do *not* need to look at the code of any other functions, and I do not need to know everything that `f()` calls transitively. Reasoning about a function then becomes an almost purely local activity. We don't have to think hard about what the rest of the program is doing.

Preconditions and postconditions also serve as useful documentation. If Bob writes down pre- and post-conditions for the module he has built, and Alice wants to invoke Bob's code, she only has to look at the pre- and post-conditions—she does not need to look at or understand Bob's code. This is a useful way to coordinate activity between multiple programmers: each module is assigned to one programmer, and the pre- and post-conditions become a kind of contract between caller and callee. For instance, if Alice knows she is going to have to invoke Bob's code, then when the system is designed Alice and Bob might negotate the interface between their code and the contract on who is responsible for what.

There is one more major use for this kind of reasoning. If we want to avoid security holes and program crashes, there are usually some implicit requirements the code must meet: for instance, it must not divide by zero, it must not make out-of-bounds accesses to memory, it must not dereference null pointers, and so on. We can then try to prove that our code meets these requirements using the same style of reasoning. For instance, any time a pointer is dereferenced, there is an implicit precondition that the pointer is non-null and in-bounds.

Here is an example of using this kind of reasoning to prove that array accesses are within bounds:

```
/* Requires: a != NULL and a[] holds n elements */
int sum(int a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
            /* Loop invariant: 0 <= i < n */
        total += a[i];
    return total;
}
```

In this example, the loop invariant is straightforward to establish. It is true at the entrance to the first iteration (since the first iteration ensures $\mathtt{i} = 0$), and it is true at the entrance to every subsequent iteration (since the loop termination condition ensures $\mathtt{i} < \mathtt{n}$, and since `i` only increases), so the array access `a[i]` is within bounds.

Of course, in general, proving the absence of buffer overruns might be much more difficult, depending on how the code is structured. However, if your code is structured in such a way that it is hard to provide a proof of no buffer overruns, perhaps you may wish to consider re-structuring the code to make the absence of buffer overruns more evident.

This might all look awfully tedious. The good news is that it does get a lot easier over time. With practice, you won't need to down detailed invariants before every statement; there is so much redundancy that you'll be able to derive them in your head easily. In practice, good programmers might write down the preconditions and postconditions and a loop invariant for every loop, and that will be enough to confirm that all is well. The bad news is that, even with practice, reasoning about your code still does take time and energy—however, it seems to be worth it for code that needs to be highly secure.

While we have presented this in a fairly formal way, in practice good programmers often do the same kind of reasoning without bothering with the formal notation. Also, good programmers sometimes omit the obvious parts of the invariants and write down only the parts that seem most important. Often, we think about data structures and code in terms of the invariants it ought to satisfy first, and only then write the code.

This kind of formal reasoning can be formalized very carefully using the tools of mathematical logic. In fact, there has been a lot of research into tools that use automated theorem provers to try to mathematically prove the validity of a set of alleged pre- and post-conditions (or even to help infer such invariants). You could take a whole course on the topic, but for reasons of time, we won't go any further in this course. Our goal was merely to show you enough to get started on your own, and maybe stir you to investigate further on your own.

By the way, you may have noticed how useful it is to be able to "speak mathematics" fluently. Now you know one reason why we make you take Math 55 or CS 70 as part of your computer science education.

3 Defensive Programming

Defensive programming is like defensive driving: the idea is to avoid depending on anyone else around you, so that if anyone else does something unexpected, you won't crash. Defensive programming is about surviving unexpected behavior by code, rather than by other drivers, but otherwise the principle is similar.

Software engineering normally focuses on functionality: if the code is given meaningful inputs, then it should produce useful and correct outputs. For security, we often care more about what happens when the program is given invalid, unexpected, or ridiculous inputs: the program had better not crash, cause undesirable side-effects, or produce dangerous outputs even when the inputs are nonsensical. Defensive programming involves applying this idea at every interface and every security perimeter, so that each 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 too 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 has many flaws:

- `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 slightly 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.

- *Validate all inputs.* Sanity-check all inputs from the rest of the program. Inputs that could have come from the adversary (e.g., that came from a network packet) must be treated with particular caution. Check that the input looks reasonable. It is better to conservatively limit the input to values that were expected (even if this might cause some loss of functionality in obscure unexpected cases) than to liberally allow everything (which might permit security holes in cases the programmer didn't think of).

  For instance, what's wrong with this code?

  ```
  char *username = getenv("USER");
  char *buf = malloc(strlen(username)+6);
  sprintf(buf, "mail %s", username);
  ```

```
FILE *f = popen(buf, "r");
fprintf(f, "Hi.\n");
fclose(f);
```

Answer: If the attacker controls the `USER` environment variable, then he might arrange for its value to be something like "`daw; /bin/rm -rf $HOME`", and then the above code will have very unpleasant consequences: `popen()` passes its input to the shell for execution, and the shell will then execute the command "`mail daw`" followed by the command "`/bin/rm -rf $HOME`". The solution is to validate that the username looks reasonable.[1]

- *Whitelist, don't blacklist.* A common mistake, when validating input from an untrusted source, is to try to enumerate bad inputs and block them. Don't do that. That is known as *blacklisting*, and it is analogous to a default-allow policy. You will inevitably overlook some patterns of dangerous inputs. Instead, use a *whitelist* of known-good types of inputs, and block anything else. You'll recognize this as an instance of a default-deny policy, which is much safer.

  For instance, if you are given a username supplied by the attacker, you might check that it matches the regular expression `[a-z][a-z0-9]*` before proceeding. A sample implementation:

```
char *validate_username(char *u) {
    char *p;
    if (!u || *u < 'a' || *u > 'z')
        die();
    for (p=u+1; *p; p++)
        if ((*p < '0' || *p > '9') && (*p < 'a' || *p > 'z'))
            die();
    return u;
}
```

- *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 may wish to avoid leaking memory or other resources, since enough memory is leaked the program might cease to operate usefully. You may also want to defend against algorithm 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 that uses a hashtable with $O(1)$ expected time per lookup, but $O(n)$ worst-case time, the attacker might send packets that trigger the $O(n)$ worst-case behavior and cause the program to 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 might wish to verify that integer overflow is impossible.

[1] Another problem is that, if the attacker can control other environment variables (e.g., `PATH`), then he can cause the wrong `mail` command to be invoked. For instance, the attacker might arrange for a malicious binary to be installed as `/tmp/mail`, set `PATH` to include `/tmp`, and then somehow cause the above code sequence to be executed. This requires validating the whole environment.

- *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 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.

So far we have considered very simple cases where we only have to think about a single client. More generally, suppose we are writing a module *M* that provides some functionality to multiple *clients*, who each call *M* to benefit from its functionality. It is important for *M* to defend itself against malicious clients. It is also important for *M* to ensure that one malicious client cannot disrupt other clients. Thus, when *M* is performing some function on behalf of a client, there are two cases:

- *If a well-behaved client supplies valid inputs, M must provide correct and useful results.* When *M* is invoked with a valid and meaningful request, *M* must respond correctly. This is primarily a functionality requirement. It may also be relevant to security, because the client may be relying upon *M* to do its job correctly.

- *If a misbehaving client supplies invalid inputs, M does not need to provide useful service to this client, but other clients must not be disrupted.* When *M* is invoked with meaningless, unexpected, or malicious input, there is no requirement that *M* provide a useful response to this client. However, *M* must protect itself from such requests, and *M* must not allow its internal state to become corrupted or harmful side effects to occur. *M* must maintain the consistency of its internal data structures no matter what inputs it receives. Also, *M* must ensure that other clients are not disrupted by requests from a malicious client, and that all well-behaved clients contain to receive correct and useful results.

Of course, *M* might in turn invoke other utility modules, relying upon them, so that *M* is itself a client of those other modules. The same requirements will apply.

There is a special case where we do not have to worry about multiple clients. Suppose *M* computes a pure function, with no internal state and performing no I/O, so that its output depends deterministically on its input. In this case, we do not need to worry about one client disrupting another client or corrupting *M*'s state. Thus, functional programming can simplify the task of defensive programming.

How does defensive programming relate to the use of preconditions? 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, 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.

4 Selecting a Programming Language

If you are lucky, you may have the opportunity to choose a programming language, libraries, operating system, or development environment. Here is some advice for how to make a choice that is best for security.

- *Pick tools that you know well.* The most important advice is to know your tools well. Many security bugs are caused by insufficient familiarity with obscure corner cases in the language, libraries, or programming environment. It takes a long time to gain experience with a language and programming environment; thus, it makes sense to amortize this investment by choosing tools that you know extremely well. A good test of how well you know your language is: Have you read the formal language specification? Do you understand everything in there?

- *Pick a programming platform designed for safety.* A significant fraction (over 50%, by some measures) of security holes in C code are related to absence of bounds-checking in C. If you have the chance to pick a language that provides automatic bounds-checking for all arrays and pointers, automatic detection of memory management errors, automatic detection of errors with uninitialized variables, and so on, then it is often helpful to do so. Also, type checking can be a powerful tool to reduce the incidence of certain kinds of programming bugs; you might choose a language that provides strong support for types. Similar comments apply to the choice of libraries and the rest of the programming platform.

  For instance, assembly language is generally a poor choice for general-purpose programming, because it is so easy to make a devastating mistake. These days, it is widely recognized that one should use assembly language only when it is absolutely necessary and only very sparingly, if at all. Many security researchers are starting to feel the same way about languages like C and C++. Type-safe languages—e.g., Java, C#, Ada, or ML—have many advantages from a security standpoint.

Of course, you will not always have the opportunity to choose the language on the basis of what is best for security. For instance, other considerations may dominate, or you may be forced to maintain legacy code. A corollary of the above comments is that if you are programming in a language that isn't optimal for security, you need to be extra careful. If you don't know the language extremely well, it would be good to try to learn it better, and to avoid the more obscure corners of the language and stick to the core that you know best. If you are forced to program in a language without automatic bounds-checking, extra caution is warranted. You may want to force yourself to stick a rigid discipline where you insert a manual bounds-check anywhere any array or pointer operation is performed, or you may wish to write your code in a way so you can prove (using the formal reasoning methods outlined earlier) that out-of-bounds accesses are impossible.

Here is some advice specific to C programming:

- *Avoid buffer overruns.* Prove that no array access, pointer dereference, or structure access can overflow the bounds of the associated object. Make all preconditions, loop invariants, and object invariants that are needed to prove this explicit in the code (e.g., as comments).

- *Avoid undefined behavior.* If you read the C standard, you will discover the special term *undefined behavior* used frequently. There are many language primitives that have their own implicit precondition that must be met. If the precondition is not met, undefined behavior results, which means that the compiler is allowed to cause anything at all to happen. For instance, evaluating `a[i]` triggers undefined behavior if the array index `i` is out of bounds; so, too, does dividing by zero, or causing any number of other error conditions. In these cases, no promises are made, and anything might happen. Buffer overruns are one special kind of undefined behavior, and in practice, undefined behavior may mean that an attacker can hijack control of the program. Thus, you should be careful to ensure that your code can never invoke undefined behavior.

- *Get familiar with the C standard.* The official C standard is the definitive specification of what is guaranteed about the behavior of the language and the libraries. You should have a copy on your bookshelf or bookmarked in your web browser. Even the best textbooks, man pages, and informal guides occasionally get things slightly wrong[2].

5 Process

Security is an ongoing process, and security must be integrated into all phases of the system development lifecycle: requirements analysis, design, implementation, testing, quality assurance, bugfixing, and maintenance.

- *Test code thoroughly before deployment.* Testing can help eliminate bugs. It is worth putting some 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 (integers are 0, 1, -1, $2^{31}-1$, -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

[2] For instance, here is one example I recently learned about. One well-regarded C reference book states that `toupper()` can be safely called on any argument of type `char`. This is wrong. Instead, as the C standard correctly says, it can be called safely on arguments of type `unsigned char` and on the value `EOF`, but anything else may lead to undefined behavior. Passing `toupper()` an argument of type `char` can lead to an out-of-bounds access and undefined behavior, on platforms where `char` is a signed type. The reference book's description was only slightly wrong, but it was wrong enough that it could have led to a security hole.

to make the same erroneous assumption when reviewing her own code as when she wrote it, while someone else may spot the error 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.