CS 161 Computer Security
Dawn Song
Fall 2012
Optional Notes

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. A *precondition* for `f()` is an assertion (a logical proposition) that must hold for the inputs supplied to `f()`. The function `f()` is presumed 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 ensure that these preconditions hold. 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 `p` $\neq$ `NULL` must hold at the entrance to `deref()`. As long as all callers ensure this precondition, it will be safe to call `deref()`.[1]

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 &&
    size(a) >= n &&
    for all j in 0..n-1,  a[j] != NULL */
int sum(int *a[], size_t n) {
```

Material courtesy Prof. David Wagner

[1]Technically speaking, we also need to know that `p` is a valid pointer: i.e., it is safe to dereference. To be strictly correct, we ought to add that to the precondition. However, here we will follow the convention that every pointer-typed variable is implicitly assumed to have, as an invariant condition, that it is either `NULL` or valid. This convention simplifies and shortens preconditions, postconditions, and invariants.

```
    int total = 0;
    size_t i;
    for (i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

The third part of the 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 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 at the level of a single line of code. 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 preceding statement, and a precondition for the next statement.

It is pretty straightforward to tell whether a statement in isolation meets 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 and you use a proof by induction to prove it valid.

et'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 the input must be at least 1 for this implementation to be correct[2]. Suppose we want to prove that the value of `fact(.)` is always positive. We'll annotate the code with invariants (in blue):

[2]Another prerequisite is that the input cannot be too large, to avoid integer overflow. However we shall ignore integer overflow in this example, to simplify the exposition.

```
/* 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[3].

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)
```

[3]Again, ignoring integer overflow (which ruins everything, once `n` gets large enough).

```
        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 $\texttt{n} \geq 1$ (by the precondition), that $\texttt{n} \neq 1$ (since the if statement didn't follow its then branch), and that `n` is an integer. It follows that $\texttt{n} \geq 2$, or that $\texttt{n} - 1 \geq 1$. 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[4].

Let's try an example. Here is some code that prints out the decimal representation of an integer, but reversed (least significant digit first):

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";
    while (n != 0) {
        int d = n % 10;
        putchar(digits[d]);
        n = n / 10;
    }
    putchar('0');
}
```

[4]As a reminder, again we have ignored integer overflow—which is not valid.

A prerequisite is that the input `n` must be non-negative for this function to work correctly, hence the precondition. Suppose we want to prove that the array dereference (`digits[d]`) never goes outside the bounds of the array. We'll annotate the code with invariants (in blue):

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";     /* n >= 0 */
    while (n != 0) {                  /* n > 0 */
        int d = n % 10;               /* 0 <= d && d < 10 && n > 0*/
        putchar(digits[d]);           /* 0 <= d && d < 10 && n > 0*/
        n = n / 10;                   /* 0 <= d && d < 10 && n >= 0*/
    }
    putchar('0');
}
```

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. For instance, the function's precondition implies the invariant after the first line of the function body. Also, we can prove by induction that `n` > 0 is a loop invariant: we know that `n` ≥ 0 holds at the entry to the loop and at the end of the prior iteration; if we enter the loop, then we also know `n` $\neq 0$; and these two, taken together, imply `n` > 0. Tracing forward, we can see that if `n` > 0 holds at the beginning of the loop body, then `n` ≥ 0 holds at the end of the loop body. The validity of the loop invariant follows by induction on the number of iterations of the loop. The conclusion is that the array accesses in `binpr()` will always be in-bounds, as long as `binpr()`'s precondition is met.

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

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";
    if (n == 0) {
        putchar('0');
        return;
    }
    int d = n % 10;
    putchar(digits[d]);
    int m = n / 10;
    binpr(m);
}
```

Do you see how to prove that the array accesses are always valid? Let's do it:

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";
    if (n == 0) {
        putchar('0');
        return;
    }                    /* n > 0 */
    int d = n % 10;      /* n > 0 && 0 <= d && d < 10 */
    putchar(digits[d]);  /* n > 0 && 0 <= d && d < 10 */
    int m = n / 10;      /* n > 0 && 0 <= d && d < 10 && m >= 0 */
    binpr(m);
}
```

Before the recursive call to `binpr()`, we know that $m \geq 0$ (by the annotations). That's very good, because it means the precondition is met when making the recursive call. As a result, we're entitled to conclude that `binpr(m)` is safe (does not perform any out-of-bounds array access). Also, we can easily see that the expression `digits[d]` is safe, by virtue of the annotations we've filled in. It follows that, as long as the precondition to `binpr()` is respected, the function is memory-safe.

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 we can verify function `f()` by looking only at the code of `f()` and the annotations on every function that `f()` calls—but we do *not* need to look at the code of any other functions, and we 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 negotiate 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 && size(a) >= n */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* Loop invariant: 0 <= i && i < n && n <= size(a) */
        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 during the first iteration, `i` = 0), and it is true at the entrance to every subsequent iteration (since the loop termination condition ensures `i` < `n`, and since `i` only increases, and since `n` and `size(a)` never change), so the array access `a[i]` is always within bounds.

Of course, proving the absence of buffer overruns in general 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 should 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 write 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, you 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, you can do the same kind of reasoning without bothering with the formal notation. Also, you can often omit the obvious parts of the invariants and write down only the parts that seem most important. Sometimes, it is helpful to 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 reasoning can be formalized more precisely 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 CS 161. Your basic intuition should be enough to handle most cases 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.

1 Optional: More on Defensive Programming

For those who are interested in secure coding and defensive programming, here is a little bit of additional information on the concept (purely optional). 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.

ne could rewrite it as follows:

```
char *double(char *str) {
    size_t len;
    if (p == NULL)
        return NULL;
    len = strlen(str);
    if (len > (SIZE_MAX-1)/2)
        return NULL;
    char *p = malloc(2*len+1);
    if (p == NULL)
        return NULL;
    strcpy(p, str);
    strcpy(p+len, str);
    return p;
}
```

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.

he above code might be re-written as something like:

```
char *username = validate_username(getenv("USER"));
char *argv[] = {username, NULL};
char *safeenv[] = {"PATH=/bin:/usr/bin", NULL};
execve("/bin/mail", username, safeenv);
...
```

Of course, one would need to add appropriate error-checking, but I hope this conveys the essence.

- *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 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 that uses a hash table 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 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 might wish to verify 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 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? 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.

2 Optional: Security Throughout the Software Development Process

For those who are interested in security as it applies throughout the software development lifecycle, here is a little bit of additional information on the concept (purely optional).

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, bugfixing, and maintenance. Security is not a feature or checklist item that can be bolted-on after the software has been developed.

- *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 (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 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.