CS 161 Computer Security
Spring 2010 Paxson/Wagner Notes 1/25

Defending Against Memory-Safety Vulnerabilities

Last lecture, we saw a class of attacks on programs, based upon exploiting memory-safety violations. This lecture, let's look at some techniques available to defend against memory-safety vulnerabilities. There are at least five:

- **Secure coding practices.** We can adopt a disciplined style of programming that avoids these bugs.
- **Better languages/libraries.** We can adopt languages or libraries that make such mistakes harder to commit.
- **Runtime checking.** We can have our compiler (or other tools) automatically inject runtime checks everywhere they might be needed to detect and prevent exploitation of memory-safety bugs, so that if our code does have a memory-safety bug, it won't be exploitable by attackers.
- **Static analysis.** We can use a compiler (or a special tool) to scan the source code and identify potential memory-safety bugs in the code, and then task developers with fixing those bugs.
- **Testing.** We can applying testing techniques to try to detect memory-safety bugs in the code, so that we can fix them before the software ships to our customers.

Many of these techniques can be applied to all kinds of security bugs, but for concreteness, let's explore how they can be applied to protect against memory-safety vulnerabilities.

1 Secure Coding Practices

In general, before performing any potentially unsafe operation, we can write some code to check (at runtime) whether the operation is safe to perform and abort if not. For instance, instead of

```
char digit_to_char(int i) { // BAD
    char convert[] = "0123456789";
    return convert[i];
}
```

we can write

```
char digit_to_char(int i) { // BETTER
    char convert[] = "0123456789";
    if (i < 0 || i > 9)
        return "?"; // or, call exit()
    return convert[i];
}
```

This code ensures that the array access will be within bounds. Similarly, when calling library functions, we can use a library function that incorporates these kinds of checks, rather than one that does not. Instead of

```
char buf[512];
strcpy(buf, src); // BAD
```

we can write

```
char buf[512];
strlcpy(buf, src, sizeof buf); // BETTER
```

The latter is better, because `strlcpy(d,s,n)` takes care to avoid writing more than `n` bytes into the buffer `d`. As another example, instead of

```
char buf[512];
sprintf(buf, src); // BAD
```

we can write

```
char buf[512];
snprintf(buf, sizeof buf, src); // BETTER
```

Instead of using `gets()`, we can use `fgets()`. And so on.

In general, we can check (or otherwise ensure) that array indices are in-bounds before using them, that pointers are non-null and in-bounds before dereferencing them, that integer addition and multiplication won't overflow or wrap around before performing the operation, that integer subtraction won't underflow before performing it, that objects haven't already been de-allocated before freeing them, and that memory is initialized before being used.

We can also follow *defensive programming* practices. 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 other code, rather than by other drivers, but otherwise the principle is similar.

Defensive programming means that each module takes responsibility for checking the validity of all inputs sent to it. Even if you "know" that your callers will never send you a NULL pointer, you check for NULL anyway, just in case, because sometimes what you "know" isn't actually true, and even if it is true today, it might not be true tomorrow as the code evolves. Defensive programming is about minimizing trust in the other components your code interacts with, so that the program fails gracefully if one of your assumptions turns out to be incorrect. It's usually better to throw an exception or even stop the program than to allow a malicious code injection attack to succeed. Don't write fragile code; strive for robustness. Defensive programming might lead to duplicating some checks or introducing some unnecessary checks, but this may be a price worth paying for reducing the likelihood of catastrophic security vulnerabilities.

It can also be helpful to perform code reviews, where one programmer reviews the code written by another programmer to ensure (among other things) that the code follows secure code practices. For instance, some companies have a requirement that all code be reviewed by another programmer before being checked in. To help code reviewers (and yourself), a good principle is to organize your code so that it is obviously correct. If the correctness of your code would not be obvious to a reviewer, rewrite it until its correctness is self-evident. As the great programmer Brian Kernighan once wrote:

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

In the case of avoiding memory-safety bugs, code should be rewritten until it is self-evident that it never performs any out-of-bounds memory access, for instance by introducing explicit runtime checks before any questionable memory access.

One potential issue with relying solely upon secure coding practices is that we are still relying upon programmers to never make a mistake. Programmers are human, and so errors are inevitable. The hope is that secure coding practices can reduce the frequency of such errors and can make them easier to spot in a code review or debugging session. Also, static analysis tools (see below) may be able to help detect deviations from secure coding practices, further reducing the incidence of such errors.

Many secure coding practices have a strong overlap with good software engineering principles, but the demands of security arguably 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. Software reliability normally focuses primarily on those 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 harder than dealing with mischance.

2 Better Languages and Libraries

Languages and libraries can help avoid memory-safety vulnerabilities by eliminating the opportunity for programmer mistakes. For instance, Java performs automatic bounds-checking on every array access, so programmer error cannot lead to an array bounds violation. Also, Java provides a String class with methods for many common string operations. Importantly, every method on String is memory-safe: the method itself performs all necessary runtime checks and resizes all buffers as needed to ensure there is enough space for strings. Similarly, C++ provides a safe string class; using these libraries, instead of C's standard library, reduces the likelihood of buffer overflow bugs in string manipulation code.

3 Runtime Checks

Compilers and other tools can reduce the burden on programmers by automatically introducing runtime checks at every potentially unsafe operation, so that programmers do not have to do so explicitly. For instance, there has been a great deal of research on augmenting C compilers so they automatically emit a bounds check at every array or pointer access.

One challenge is that automatic bounds-checking for C/C++ has a non-trivial performance overhead. Even after decades of research, the best techniques still slow down computationally-intensive code by 10%-150% or so, though the good news is that for I/O-bound code the overheads can be smaller[1]. Because many security-critical network servers are I/O-bound, automatic bounds-checking may be feasible for some C/C++ programs. The reason that bounds-checking is expensive for C is that one must bounds-check every pointer access, which adds several instructions for the check for each instruction that accesses a pointer, and C programs use pointers frequently. Also, to enable such checks, pointers have to carry information about their bounds, which adds overhead for the necessary book-keeping.

Another potential issue with automatic bounds-checking compilers for C/C++ has to do with legacy code. First, the source code of the program has to be available, so that the program can be re-compiled using a bounds-checking compiler. Second, it can be difficult to mix code compiled with bounds-checking enabled with libraries or legacy code not compiled in that way. To check the bounds on a pointer `p`, we need some way to know the start and end of the memory region that `p` points into, which requires either changing the memory representation of pointers (so that they are an address, a base, and an upper bound) or introducing extra data structures. Either way, this poses challenges when code compiled in this way interacts with code compiled by an older compiler.

There are other techniques that attempt to make it harder to exploit any memory-safety bugs that may existing the code. You might learn about some of them in discussion section[2].

4 Static Analysis

Static analysis is a technique for scanning source code to try to automatically detect potential bugs. You can think of static analysis as runtime checks, performed at compile time: the static analysis tool attempts to predict whether there exists any program execution under which a runtime check would fail, and if it finds any, it warns the programmer. Sophisticated techniques are needed, and those techniques are beyond the scope of this class, but they build on ideas from the compilers and programming language literature for automatic program analysis (e.g., ideas initially developed for compiler optimization).

The advantage of static analysis is that it can detect bugs proactively, at development time, so that they can be fixed before the code has been shipped. Bugs in deployed code are expensive, not only because customers don't like it when they get hacked due to a bug in your code, but also because fixes require extensive testing to ensure that the fix doesn't make things worse. Generally speaking, the earlier a bug is found, the cheaper it can be to fix, which makes static analysis tools attractive.

One challenge with static analysis tools is that they make errors. This is fundamental: detecting security bugs can be shown to be undecidable (like the halting problem), so it follows that any static analysis tool will either miss some bugs (false negatives), or falsely warn about code that is correct (false positives), or both. In practice, the effectiveness of a static analysis tool is determined by its false negative rate and false positive rate; these two can often be traded off against each other. At one extreme are verification tools, which are guaranteed to be free of false negatives: if your code does not trigger any warnings, then it is guaranteed to be free of bugs (at least, of the sort of bugs that the tool attempts to detect). In practice, most developers accept a significant rate of false negatives in exchange for finding some relevant bugs, without too many false positives.

[1]If you're interested in learning more, you can read a recent research paper on this subject at `http://www.usenix.org/events/sec09/tech/full_papers/akritidis.pdf`.

[2]For those interested in reading more, you can read about ASLR and the NX bit.

5 Testing

Another way to find security bugs proactively is by testing your code. A challenge with testing for security, as opposed for functionality, is that security is a negative property: for security, we want to prove that nothing bad happens, even in unusual circumstances; whereas standard testing focuses on ensuring that something good does happen, under normal circumstances. It is a lot easier to define test cases that reflect normal, expected inputs and check that the desired behavior does occur, then to define test cases that represent the kinds of unusual inputs an attacker might provide or to detect things that are not supposed to happen.

Generally, testing for security has two aspect:

1. *Test generation.* We need to find a way to generate test cases, so that we can run the program on those test cases.
2. *Bug detection.* We need a way to detect whether a particular test case revealed a bug in the program.

Fuzz testing is one simple form of security testing. Fuzz testing involves testing the program with random inputs and seeing if the program exhibits any sign of failure. Generally, the bug detection strategy is to check whether the program crashes (or throws an unexpected exception). For greater bug detection power, we can enable runtime checks (e.g., automatic array bounds-checking) and see whether any of the test cases triggers a failure of some runtime check. There are three different approaches to test generation that are commonly taken, during fuzz testing:

- *Random inputs.* Construct a random input file, and run the program on that input. The file is constructed by choosing a totally random sequence of bytes, with no structure.
- *Mutated inputs.* Start with a valid input file, randomly modify a few bits in the file, and run the program on the mutated input.
- *Structure-driven input generation.* Taking into account the intended format of the input, devise a program to independently "fuzz" each field of the input file. For instance, if we know that one part of the input is a string, generate random strings (of random lengths, with random characters, some of them with `%` signs to try to trigger format string bugs, some with funny Unicode characters, etc.). If another part of the input is a length, try random integers, try a very small number, try a very large number, try a negative number (or an integer whose binary representation has its high bit set).

One shortcoming of purely random inputs is that, if the input has a structured format, then it is likely that a random input file will not have the proper format and thus will be quickly rejected by the program, leaving much of the code uncovered and untested. The other two approaches address this problem. Generally speaking, mutating a corpus of valid files is easier than writing a test generation suite customized to the particular input format under consideration, but may be less effective.

Usually, fuzz testing involves generating many inputs: e.g., hundreds of thousands or millions. What it sacrifices in sophistication, it makes up for in quantity. Fuzz testing is popular in industry today because it is cheap, easy to apply, and somewhat effective at finding some kinds of bugs (more effective than you would think it has any right to be)[3]

[3]For those interested in learning more, I can recommend the following set of slides: `http://toorcon.org/2007/talks/60/real_world_fuzzing.pdf`. If you like to try things out, you could check out zzuf, a very easy to use mutation fuzzer for Linux: `http://caca.zoy.org/wiki/zzuf`. For instance, on Linux you can have some fun with a command like `zzuf -v -s 1:1000 valgrind -q --leak-check=no --error-exitcode=1 unzip -o foo.zip`; look for error messages from Valgrind, prefixed with `==NNNNN==`.

6 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()`[4].

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) {
    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

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

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:

[4]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, I 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. I hope that this convention is not confusing.

```
/* 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 $\mathtt{v} = 0$ (no matter what the precondition is). Or, if the precondition for the statement "`v=v+1;`" is $\mathtt{v} \geq 5$, then a valid postcondition would be $\mathtt{v} \geq 6$. As another example, if the precondition for the statement "`v=v+1;`" is $\mathtt{w} \leq 100$, then $\mathtt{w} \leq 100$ 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.

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');
}
```

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 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 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, 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, 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 CS161. 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.

7 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.

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.

- *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.

- *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.

8 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 (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 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.