All.Net Analyst Report and Newsletter

Welcome to our Analyst Report and Newsletter

Input checking

I recently encountered several cases where my previously operating code failed to properly check inputs. I fixed them right away, and they didn't (couldn't) actually produce any harm, but it bothered me that I screwed up this way, so I thought I would pester you about it as well.

Why don't we check inputs well enough?

I say "we" because I know it's true of me (or was true till I fixed these), and I know its true of many of the commonly used programs we encounter, because failure to adequately check inputs keeps coming up as a root cause of protection failures. I highly suspect its true of most programmers because I have seen enough code examples to believe it wasn't just that I happened across the rare cases.

I think it's because we separate input from these checks. Here's an example from perl: 'a=<IN>;' I could have picked any computer language of course: 'read(0,&a, 23);', 'read a', 'a.getText()', and on and on. But this won't be changing anytime soon, and frankly, having a built-in set of functions for inputting every one of the possible input sequences is problematic in its own way. It might be nice to have inout routines for integers, dates, real numbers, and so forth, but nothing of this sort can be universal, even if 'yacc' seems to do a good job for most regularized syntax.

A poor but working example

My current solution is rather pathetic. I have a routine that does minimal checks and I try to use it for each and every input just after I get it. Here's an example from a perl script I use:

```
sub argcheck{my $arg=$_[0];my $minlen=$_[1];my $maxlen=$_[2];my $makeup=$_[3];
   if (length($arg) < $minlen) {return 0;}    if (length($arg) > $maxlen) {return 0;}
   if ($arg =~ /^$makeup$/) {return 1;}    return 0; # default failure
}
```

This is used after an input is taken from a form provided by a user through a Web interface. It does the simplest sort of check. By example, as soon as the argument ($Org) is available:

```
if (argcheck($Org, 1, 100, "[a-zA-Z0-9]+")) {whatever} else {badinput(...);exit;}
```

This example checks minimum length (1), maximum length (100) and regular expression match (any number of letters or digits). If the input check fails, the program calls "badinput" then exits. Badinput in this case prints out whatever I want to appear on the browser screen.

I want to note that this is a really poor example of input checking, because it is a really gross level check. It does not deal with the actual syntax issues of the particular input, but rather just constrains it in some minimal ways. It does in fact eliminate all of the relevant problems that can occur with this particular piece of code because the code then checks to see if a directory exists with the same name as the input string, and if it doesn't, fails by doing the same {badinput(...);exit;} sequence. Because it excludes characters that could cause directory traverse, internal mechanisms that change the name, lengths that could cause other failures, and so forth, the only remaining matches will be existing paths explicitly provided for use.

A better example

I wish I had a better example, but unfortunately, none of the ones I have like this will fit in such a small article. By that, I mean, that the lexical analysis required to perform such checks for real languages tend to come out rather long. And it seems a bit much to expect that every programmer and every program out there will replace 'a=<IN>;' with page of syntactic description of the valid input syntax for each particular variable being input. To get a sense of this, suppose we want to input a valid future date... here is an outline:

- It has to be in a known format - let's choose YYYY-MM-DD
  - Other inputs will have to be reformatted to the extent feasible or the user will have to be pestered again and again on the invalidity of their inputs
- The maximum year is 9999 - unless we want to be more restrictive
- The maximum month is 12
- The maximum day is 31, but varies with month... and year and month
- The YYYY must be at least the current YYYY
  - Unless this is the last day of the current year in which case it has to be next year.
  - Of course this will vary by time zone (the user may be in a different one too)
- The MM must be at least the current month if the year is this year...
  - Or if this is December...
- The DD must be at least the current day of the month if the month is this month...
  - Unless this is the last day of the month or the year is not this year
    - Which varies based on year if the month is February
      - Don't forget leap centuries...
- And please note this is only the current Western calendar... there are others

So what should we really expect?

The thing that makes this seemingly simple task so complicated is that it's not really simple at all. It's complicated to do at all, more complicated to do well, and even more complicated to do 'right' or even define what that would be. And this is just a generic example for a standard problem that enormous numbers of programs and programmers deal with every day.

We should expect failure because we, as a community, haven't addressed the basics of doing input right, haven't standardized the process, and haven't built it in so it's easy to do. At a minimum, a simple way to check standard input types and fields should be built in.

Is the situation really that bad?

I decided to do some Web searching to seek out whether my assessment above is right. You should try it as well. Pick a computer language (I picked perl for my example) and search for something like "input checking perl" or use "validation" instead of "checking" if you like. You will find lots of results - Google found about 469 million in 0.37 seconds. Let's just go for date checking. I used "input checking dates perl".

The first result was really a good example of what I am talking about. It came from a really worthwhile site: 'http://www.perlmonks.org/?node_id=564594'. It started with a simple answer:

> if ($yyyymmdd !~ /(\d{4})(\d\d)(\d\d)/) {print "Bad data string provided: $yyyymmdd\n"; return 0;} my ($year, $month, $day) = ($1, $2, $3);

In English, the variable (yyyymmdd - they assumed no '-' would be included) was checked to make certain it had all digits in every field as it was split apart. The next entry noted that this would not do since there would be invalid dates (e.g., 0000/98/34). The next one used a really worthwhile trick of calling an internal routine called "timelocal" that does some internal checking, and returns failure if that internal routine returns failure However, "this won't work for valid dates less than Jan 1 1970 and greater than Jan 19 2038.". Next came this gem:

> perl -we 'use Date::Manip; sub valid_yyyymmdd { $_[0] =~ /^(\d{8})$/ and ParseDate($1) and 1; } valid_yyyymmdd($_) and print "$_\n" for qw" 20060630 20060631 20060730 20060731 20060732 2006.07.30 06.07.31";'

Again, it uses an library function for date manipulation for a different purpose, testing it with various inputs. And it self-reports: "I'll do this in two steps. First I'll check that the date is indeed eight digits, then I'll check if it refers to a valid date with the Date::Manip module (so this will fail with historical dates using the Julian calendar)."

The next response indicated 7 different articles on date parsing with regular expressions. Next there is a nice short one using an ISO8601 format checking routine, then one that programs out the various conditions for leap years etc. and uses a table for days per month for all but February, and finally another one using another library:

> use Date::Calc qw/check_date/; my @date = $yyyymmdd =~ /^(\d{4})(\d{2})(\d{2})$/; die "ERROR: bad date '$yyymmdd'" unless check_date(@date); printf "Valid (y,m,d)=(%d,%d,%d)\n", @date;

Of course none of these solve the problem I stated for checking a future date. But with a bit of effort we could, for example, compare YYYY to current, and if more than current and a valid date return "true", otherwise if less than current return false, and otherwise compare months. The month comparison does the same thing relative to the current month, and if the month is the same, compare days. In this case, the day is either more or not, since the month comparison handles the next month issue just as the year comparison handles the next year issue. And of course all of this relies on the internal routines properly handling all cases, on the Julian calendar being used, it ignores time zones and related issues, and assumes that all of the people on the forum told the truth. After implementation, I have to test all of this, integrate it into my implementation, and all of that instead of '$yyyymmdd=<IN>'

I should also mention that Perl is perhaps the most supportive of the languages I encounter in this arena. And don't imagine checking at the Web interface fixes this. It leaves a vulnerability.

Summary

In practice, the reasons we don't do input checking well is that (1) we don't know how to do it well (2) it's nonstandard, (3) it's complex, (4) it's not built in, and (5) it's a whole lot harder to do than just taking inputs and doing some simple checks, and those are a lot more effort than just taking and trusting inputs. If we want to solve the security problem, we need to solve the input problem, and to do that, we need to start changing these things about how we do input.

