Secure programmer: Validating input

Best practices for accepting user data

This article shows how to validate input -- one of the first lines of defense in any secure program.

PDF (206 KB) |

David Wheeler (dwheelerNOSPAM@dwheeler.com), Research staff member, Institute for Defense Analyses

23 October 2003

Also available in Japanese

In July, 2003, the CERT Coordination Center reported a dangerous set of vulnerabilities in Microsoft Windows' DirectX MIDI Library. The DirectX MIDI library is a low-level Windows library for playing music stored in the MIDI format. Unfortunately, this library failed to check all the data values inside MIDI files; incorrect data values for the fields "text," "copyright," or "MThd track" in a MIDI file could cause the library to fail and attackers could exploit the failure to make the system run any code they wanted. This was especially dangerous, because Internet Explorer, when it viewed a Web page with a link to a MIDI file, would automatically load the file and try to play it. The result? An attacker could simply post a Web page that when viewed would make the browsing user's computer erase all its files, send all its confidential files elsewhere by e-mail, crash, or do whatever else the attacker wanted.

Check your input

In nearly all secure programs, your first line of defense is to check every piece of data you receive. If you can keep malicious data from entering your program, or at least keep it from being processed, your program becomes much harder to attack. This is very similar to how firewalls protect computer networks from attackers; it won't prevent all attacks, but it does make a program much more resistant. This process is called checking, validating, or filtering your data.

One obvious question is, where should the checking be performed? When the data first enters the program, or later by a lower-level routine that actually uses the data? Often, it's best to check in both places; that way, if an attacker manages to slip around one defense, they'll still encounter the other. The most important rule is that all data must be checked before it's used.

Back to top

The big mistake: Looking for incorrect input

One of the biggest mistakes developers of secure programs make is to try to check for "illegal" data values. It's a mistake because attackers are quite clever; they can often think of yet another dangerous data value. Instead, determine what is *legal*, check if the data matches that definition, and reject anything that doesn't match that definition. For security it's best to be extremely conservative to start with, and allow just the data that you know is legal. After all, if you're too restrictive, users will quickly report that the program won't allow legitimate data to be entered. On the other hand, if you're too permissive, you may not find that out until after your program has been subverted.

For example, let's say that you're going to create filenames based on certain inputs from a user. You may know that allowing users to include "/" would be a bad idea, but just checking for this one character would probably be a mistake. For example, what about control characters? Would spaces be a problem? How about leading dashes (which can cause problems in poorly-written scripts)? And could certain phrases cause a problem? In most cases, if you create a list of "illegal" characters, an attacker will find a way to exploit your program. Instead, check to make sure the input matches a certain pattern that you know is safe, and reject anything not matching the pattern.

It's still a good idea to identify values you know are dangerous: you can use them to (mentally) check your validation routines. Thus, if you know that "/" is dangerous, look at your pattern to make sure it wouldn't let that character through.

Of course, all of this begs the question: what are the legal values? The answer depends, in part, on the kind of data that you're expecting. So the next few sections will describe some common kinds of data that programs expect -- and what to do about them.

Back to top

Numbers

Let's start with what would appear to be one of the easiest kinds of information to read -- numbers. If you're expecting a number, make sure your data is in number format -- typically, that means only digits, and at least one digit (you can check this using the regular expression `^[0-9]+$`). In most cases there is a minimum value (often zero) and a maximum value; if so, make sure the number is inside its legal range.

Don't depend on the lack of a minus sign to mean that there are no negative numbers. Many number-reading routines, if presented with an excessively large number, will "roll over" the value into a negative number. In fact, a clever attack against Sendmail was based on this insight. Sendmail checked that "debug flag" values weren't larger than the legal value, but it didn't check if the number was negative. Sendmail developers presumed that since they didn't allow minus signs, there was no need to check. The problem is that the number-reading routines took numbers larger than 2^31, such as 4,294,967,269, and converted them into negative numbers. Attackers could then use this fact to overwrite critically important data and take control of Sendmail.

If you're reading a floating point number, there are other concerns. Many routines designed to read floats may allow values such as "NaN" (not a number). This can really confuse later processing routines, because any comparison with them is false (in particular, NaN is not equal to NaN!). Standard IEEE floating point has other oddities that you need to be prepared for, such as positive and negative infinities, and negative zero (as well as positive zero). Any input data that your program isn't prepared for may be exploitable later.

Back to top

Strings

Again, with strings you need to identify what is legal, and reject any other string. Often the easiest tool for specifying legal strings are regular expressions: Just write a pattern using a regular expression to describes what string values are legal, and throw away data that doesn't match the pattern. For example, `^[A-Za-z0-9]+$` specifies that the string must be at least one character long and that it can only include upper-case letters, lower-case letters, and the digits 0 through 9 (in any order). You can use regular expressions to limit which characters are allowed and to be more specific (for example, you can often limit even further what the first character can be). Just about all languages have libraries that implement regular expressions; Perl is based on regular expressions, and for C, the functions `regcomp(3)` and `regexec(3)` are part of the POSIX.2 standard and are widely available.

If you use regular expressions, be sure to indicate that you want to match the beginning (usually symbolized by `^`) and end (usually symbolized by `$`) of the data in your match. If you forget to include `^` or `$`, an attacker could include legal text inside their attack to bypass your check. If you're using Perl and you use its multi-line option (`m`), watch out: you must use `\A` for the beginning and `\Z` for the end instead, because the multi-line option changes the meaning of `^` and `$`.

The biggest problem is figuring out exactly what should be legal in the string. In general, you should be as restrictive as possible. There are a large number of characters that can cause special problems; where possible, you don't want to allow characters that have a special meaning to the program internals or the eventual output. That turns out to be really difficult, because so many characters can cause problems in some cases.

Here is a partial list of the kinds of characters that often cause trouble:

- **Normal control characters (characters with values less than 32):** This especially includes character 0, traditionally called NUL; I call it NIL to distinguish it from C's NULL pointer. NIL marks the end of strings in C; even if you don't use C directly, many libraries call C routines indirectly and can get confused if given NIL. Another problem is line ending characters, which can be interpreted as command endings. Unfortunately, there are several line ending encodings: UNIX-based systems use character linefeed (0x0a), but DOS based systems (including Windows) use the CP/M marking carriage-return linefeed (0x0d 0x0a), the Apple MacOS uses carriage return (0x0d), many IBM mainframes (like OS/390) uses next line (0x85), and some programs even (incorrectly) use the reverse CP/M marking (0x0a 0x0d).
- **Characters with values higher than 127**: These are used for international characters, but the problem is that they can have many possible meanings, and you need to make sure that they're properly interpreted. Often these are UTF-8 encoded characters, which has its own complications; see the UTF-8 discussion later in this article.
- **Metacharacters:** Metacharacters are characters that have special meanings to programs or libraries you depend on, such as the command shell or SQL.
- **Characters that have a special meaning in your program:** For instance, characters used as delimiters. Many programs store data in text files, and separate the data fields with commas, tabs, or colons; you'll need to reject or encode user data with those

values. Today, a common problem is the less-than sign (<), because XML and HTML use this.

This isn't an exhaustive list, and you often must accept some of these characters. Later articles will discuss how to deal with these characters, if you must accept them. The point of this list is to convince you to try to accept as few characters as possible, and to think carefully before accepting another. The fewer characters you accept, the more difficult you make it for an attacker.

Back to top

More specific data types

Of course, there are many more specific types of data. Here are a few guidelines for some of them.

Filenames

If the data is a filename (or will be used to create one), be very restrictive. Ideally, don't let users choose filenames, and if that won't work, limit the characters to small patterns such as `^[A-Za-z0-9][A-Za-z0-9._\-]*$`. You should consider omitting from the legal patterns characters like "/", control characters (especially newline), and a leading "." (which are hidden files in UNIX/Linux). A leading "-" is also a bad idea, since poorly-written scripts may misinterpret those as options: if there's a file named "-rf", then in UNIX/Linux the command `rm *` will become `rm -rf *`. Omitting "../" from the pattern is a good idea, to keep attackers from "escaping" the directory. When possible, don't allow globbing (selecting groups of files using the characters *, ?, [], and {}); an attacker can make some systems grind to a halt by creating ridiculously convoluted globbing patterns.

Windows has an additional problem: some filenames (ignoring the extension and upper/lower case) are always considered physical devices. For example, a program that tries to open "COM1" or even "com1.txt" in any directory will get stuck trying to talk to a serial connector. Since I'm concentrating on UNIX-like systems, I won't go into the details of how to deal with this problem, but it's worth noting, because this is an example where simply checking for legal characters isn't enough.

Locale

Given today's global economy, many programs must let each user select the language to be displayed and other culture-specific information (such as number formatting and character encoding). Programs get this information as a "locale" value provided by the user. For example, the locale "en_US.UTF-8" states that the locale uses the English language, using United States conventions, and uses UTF-8 character encoding. Local UNIX-like programs get this information from an environment variable (usually LC_ALL, but it might be set by the more specific LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, and LC_TIME; other values to check are NLSPATH, LANGUAGE, LANG, and LINGUAS). Web applications can get this through the Accept-Language request header, though other ways are often used, too.

Since the user may be an attacker, we need to validate the locale value. I recommend that you make sure that locales match this pattern:

```
^[A-Za-z][A-Za-z0-9_,+@\-\.=]*$
```

How I created this validation pattern may be even more instructive than the pattern itself. I first searched for the relevant standards and library documentation to determine what a *correct* locale should look like. In this case, there are competing standards, so I had to make sure the final pattern would accept all of them. It didn't take long to realize that only the characters listed above are needed, and limiting the character set (especially the first character) would eliminate many problems. I then thought about common dangerous characters (such as "/" for a directory separator, the lone ".." for "upper directory," leading dashes, or the empty locale), and confirmed that they wouldn't get through the filter.

UTF-8

Internationalization has had another impact on programs: character encodings. Handling text requires that there be some convention for converting characters into the numbers computers actually handle; these conventions are called *character encodings*. A particularly common way of encoding text today is UTF-8, a wonderful character encoding that is able to represent any character in essentially any language. UTF-8 is especially nice because its design makes ordinary ASCII text a simple subset of UTF-8. As a result, programs originally designed to only handle ASCII can often be easily upgraded to process UTF-8; in some cases they don't have to be modified at all.

But, like all good things, the UTF-8 design has a downside. Some UTF-8 characters are normally represented in one byte, others in two, others in three, and so on, and programs are supposed to always generate the shortest possible representation. However, many UTF-8 readers will accept "excessively long" sequences; for example, certain three-byte sequences may be interpreted as a character that's supposed to be represented by two. Attackers can use this fact to "slip through" data validators to attack programs. Your filter might not allow the hexadecimal values 2F 2E 2E 2F ("/../"), but if it allows the UTF-8 hexadecimal values 2F C0 AE 2E 2F, the program might also interpret that as "/../". So, if you accept UTF-8 text, you need to make sure that every character uses the shortest possible UTF-8 encoding (and reject any text not in its shortest form). Many languages have utilities to do this, and it's not hard to write your own. Note that the sequence "C0 80" is an overlong sequence that can represent NIL (character 00); some languages (such as Java) enshrine this particular sequence as acceptable.

E-mail addresses

Many programs must accept e-mail addresses, but correctly handling all possible legal e-mail addresses (as specified in RFC 2822 and 822) is surprisingly difficult. Jeffrey Friedl's "short" regular expression to check them is 4,724 characters long, and even that doesn't cover some cases. However, most programs can be quite strict and only accept a very limited subset of e-mails to work well. In most cases, it's okay to reject technically valid addresses like "John Doe <john.doe@somewhere.com>" as long as the program can accept normal Internet addresses in the "name@domain" format (like "john.doe@somewhere.com"). Viega and Messier's 2003 book has a subroutine that can do this check.

Cookies

Web applications often use cookie values for important data. As I'll discuss later, it's important to remember that users can reset cookie values and form data to anything they want. However, there's one important validation trick that's worth mentioning now. If you accept a cookie value, check that its domain value is what you expect (i.e., one of your sites). Otherwise, a (possibly cracked) related site might be able to insert spoofed cookies. Details on how this attack works are explained in IETF RFC 2965, if you're curious (see Resources for a link).

HTML

Sometimes your program will take data from an untrusted user and give it to another user. If the second user's program might be harmed by that data, then it's your job to protect the second user! Attacks that exploit an apparently trustworthy intermediary to pass on malicious data are called "cross-site malicious content" attacks.

These problems are especially a problem for Web applications, such as those that implement community "bulletin boards" to allow users to add running commentary. In this case, attackers can try to add commentary in HTML format with malicious scripts, image tags, and so on; their goal is to cause all other users' browsers to run the malicious code when they view the text. Since attackers are usually trying to add malicious scripts, this particular variation is called a "cross-site scripting attack" (XSS attack).

It's often best to prevent this attack by validating any HTML you accept to make sure that it doesn't have this kind of malicious content. Again, what you do is enumerate what you know is safe, and then forbid anything else.

Generally, in HTML you can at least accept these, as well as all their ending tags:

- <p> (paragraph)
- <b> (bold)
- <i> (italics)
- <em> (emphasis)
- <strong> (strong emphasis)
- <pre> (preformatted text)
- <br> (forced line break -- note that it doesn't require a closing tag)

Remember that HTML tags are not case sensitive. Don't accept any attributes unless you've checked the attribute type and its value; there are many attributes that support things such as Javascript that can cause trouble for your users.

You can certainly expand the set, but be careful. Be especially wary of any tag that causes the user to immediately load another file, such as the image tag -- those tags are perfect for XSS attacks.

One additional problem is that you'll need to make sure that an attacker can't mess up the formatting of the rest of the document, in particular, you want to make sure that any commentary or fragment doesn't look like it's "official" content. One way to do this is to make sure that any XML or HTML commands are properly balanced (anything opened is closed).

In XML, this is termed "well-formed" data. If you're accepting standard HTML, you should probably not require this for paragraph markers (<p>), because they're often not balanced.

In many cases you'll want to accept <a> (hyperlink), and for that you'll probably want to require the attribute "href". If you must, you must, but you'll need to validate the URI/URL that you're linking to -- which is our next topic.

URI/URLs

Technically, a hypertext link can be any "uniform resource identifier" (URI), and today most people only see a particular kind of URI called a "Uniform Resource Locator" (URL). Many users will blindly click on a hypertext link to a URI, under the presumption that it won't hurt to display it. Your job, as a developer, is to make sure that this user expectation is true.

Although URIs provide a lot of flexibility, if you're accepting a URI from a potential attacker, you need to check it before passing it on to anyone else. Attackers can slip lots of odd things into URIs that can fool users. For example, attackers can include queries that may cause the user to do undesirable things, and they can fool the user into thinking they're viewing a different site than what they're really viewing.

Unfortunately, it's difficult to give a single pattern that protects users in all situations. However, a mostly safe pattern that prevents most attacks, and yet lets most useful links get through (say on a public Web site), is:

```
^(http|ftp|https)://[-A-Za-z0-9._/]+$
```

A pattern that allows some more complex patterns is:

```
^(http|ftp|https)://[-A-Za-z0-9._]+(\/([A-Za-z0-9\-
\_\.\!\~\*\'\(\)\%\?]+))*/?$
```

If your needs are more complex, you'll need more complex patterns for checking the data; see my book (listed in Resources) for some alternatives.

Data Files

Complex data files and data structures are generally made up of a lot of smaller components. Simply break the file or structure down and check each piece. If you depend on certain relationships between the components, check those too. Initially, writing this code can be a little dreary, but it also has a real advantage in reliability: many mysterious problems instantly disappear if you immediately reject bad data.

Back to top

Wrap-up

Obviously, there's a lot of different kinds of data to check. But where does this data get into your program? The answer is from a surprising number of places; in fact, your program may be getting data from an attacker in ways you weren't prepared for. I'll discuss this in my next installment.

Resources

- Read David's first installment in the Secure programmer series, "Developing secure programs."
- David's book *Secure Programming for Linux and Unix HOWTO* (Wheeler, 2003) gives a detailed account on how to develop secure software.
- CERT Advisory CA-2003-18 Integer Overflows in Microsoft Windows DirectX MIDI Library describes the MIDI library vulnerability and has links to more details.
- Jeffrey Friedl's *Mastering Regular Expressions* (O'Reilly & Associates, 1997) is a good book on how to create regular expressions.
- [RHSA-2000:057-04] glibc vulnerabilities in ld.so, locale and gettext describes how a local user could raise their privileges by exploiting an error in checking the locale.
- Matt Bishop's How Attackers Break Programs, and How To Write Programs More Securely is a set of slides from SANS 2002 on how to write secure programs; slides 64-66 discuss the integer overflow exploit in Sendmail.
- John Viega and Matt Messier's *Secure Programming Cookbook* (O'Reilly & Associates, 2003) has lots of code fragments that can be useful for validating data.
- IETF Request for Comment (RFC) 2965, HTTP State Management Mechanism by Kristol and Montulli, discusses some of the security issues with Web cookies.
- "Practical Linux security" outlines a several ways to keep user accounts safe.
- Software security principles" discusses the most important things to keep in mind when designing and building secure systems.
- "Building secure software" is a two-part series that focuses on selecting the technologies that help you create secure code. Part 1 covers choices in programming languages and distributed object platforms, and Part 2 covers operating systems and authentication technologies.
- For information on what IBM is doing in the are of security, visit the IBM Research Security group home page.
- "Secure Internet applications on the AS/400 system" gives an overview of SSL and covers those OS/400 applications that use SSL.
- The IBM Linux Technology Center supports a number of security-related projects on Linux, including Linux Security Modules, the GCC extension for protecting applications from stack-smashing attacks, and more.
- "Enterprise Security for Linux" is a white paper offering a technical discussion of IBM Tivoli Access Manager for Linux. You'll find more Tivoli information on *developerWorks*'Tivoli Developer Domain.
- You'll find more Linux articles in the *developerWorks* Linux zone.