Secure programmer: Keep an eye on inputs

Find and secure the gateways into your program

David Wheeler (dwheelerNOSPAM@dwheeler.com), Research Staff Member, Institute for Defense Analyses

Summary: This article discusses various ways data gets into your program, emphasizing how to deal appropriately with them; you might not even know about them all! It first discusses how to design your program to limit the ways data can get into your program, and how your design influences what is an input. It then discusses various input channels and what to do about them, including environment variables, files, file descriptors, the command line, the graphical user interface (GUI), network data, and miscellaneous inputs.

Date: 19 Dec 2003
Level: Intermediate
Also available in: Japanese

In early 2001, many large companies installed the application "SAP R/3 Web Application Server demo," unaware that it included a dangerous vulnerability. The application included a program named saposcol, which failed to protect itself against malicious input values. An attacker could set the `PATH` environment variable to change where saposcol looked for other programs, and then create a malicious "expand" program for saposcol to run. Since saposcol had setuid root privileges, this meant that local users could quickly gain control over the entire computer system (as root), because of a single programming mistake (see Resources for a link to more on this and other incidents mentioned in this article).

The previous installment in this column identified some common input data types and how to check them. But knowing how to check data types isn't enough if you don't know where all of your data comes from. This article discusses various ways that data gets into your program -- some of which aren't obvious -- and emphasizes how to deal appropriately with them.

If you're not in control, your attacker is

The first line of defense in a secure program is to check every untrusted input. But what does that mean? It comes down to three things:

1. *Limit the portions of your program that are exposed.* If your program is subdivided into pieces -- and this is often a good idea -- try to design it so that an attacker can't communicate at all to most pieces. That includes being unable to exploit the communication paths between the pieces. It's best if attackers cannot view, modify, or insert their own data into those communication paths (including slipping in as a middleman between the pieces). If that's not possible -- such as when the pieces

communicate using a network -- use mechanisms such as encryption to counter attackers. Later articles will discuss this in more detail.
2. *Limit the types of inputs allowed by the exposed portions.* Sometimes you can change your design so that only a few inputs are even possible -- if you can, do so.
3. *Ruthlessly check the untrusted inputs.* A truly "secure" program would have no inputs, but that program would be useless. Thus, you need to ruthlessly check data on all input paths into your program from untrusted sources. The previous installment discussed how to check different types of data; this article will help you identify where that data comes from. That doesn't mean you *only* check data just entering into your program. It's often wise to check data in multiple places, but you must check all data at least once, and it's often wise at least to have a check when the data first comes in.

Back to top

It all depends on the type of your program

You must check all untrusted inputs -- but what are they? Some of them depend on what your program does. If your program is a viewer or editor of data -- such as a word processor or image displayer -- that data might be from an attacker, so it's an untrusted input. If your program responds to requests over a network, those requests might very well come from an attacker -- so the network connection is an untrusted input.

Another important factor is how your program is designed. If parts of your program run as "root" or some other privileged user, or have privileged access to data (such as the data in a database), then inputs to those parts from the unprivileged parts and programs are untrusted.

An especially important case is any program that is "setuid" or "setgid." Just running a setuid/setgid program turns on special privileges, and these programs are especially hard to make secure. Why? Because setuid/setgid programs have an especially large set of inputs -- many of them surprising -- that can be controlled by an attacker.

Back to top

Common input sources

The following sections discuss some of the common inputs and what to do about them. You should consider each of these inputs when you're writing your program, and if they are untrusted, carefully filter them.

Environment variables

Environment variables can be incredibly dangerous, especially for setuid/setgid programs and the programs they call. Three factors make them so dangerous:

1. Many libraries and programs are controlled by environment variables in ways that are incredibly obscure -- in fact, many are completely undocumented. The command shell /bin/sh uses environment variables such as `PATH` and `IFS`, the program loader ld.so (/lib/ld-linux.so.2) uses environment variables such as `LD_LIBRARY_PATH` and `LD_PRELOAD`, lots of programs use the environment variables `TERM`, `HOME`, and `SHELL` -- and *all* of these environment variables have been used to exploit programs. There are a

huge number of these environment variables; many of them are recondite variables intended for debugging, and it's pointless to try to list them all. In fact, you can't know them all, since some aren't even documented.

2. Environment variables are inherited. If program A calls B, which calls C, which calls D, then D will receive the environment variables that A received unless some program changes things along the way. This means that if A is a secure program, and the developer of D adds an undocumented environment variable that helps in debugging, that addition to D might create a vulnerability in A! This inheritance isn't accidental -- it's what makes environment variables useful -- but it also makes them a serious security problem.
3. Environment variables can be *completely* controlled by locally running attackers, and attackers can exploit this in surprising ways. As described by the environ(5) man page (see Resources), environment variables are internally stored as an array of character pointers (the array is terminated by a NULL pointer), and each character pointer points to a NIL-terminated string value of the form `NAME=value` (where `NAME` is the name of the environment variable). Why is this detail important? It's because attackers can do weird things such as create multiple values for the same environment variable name (like two different `LD_LIBRARY_PATH` values). This can easily lead libraries using the environment variables to do unexpected things, which may be exploitable. The GNU glibc library has routines that work to counter this, but other libraries and any routines that walk the environment variable list can get into trouble in a hurry.

In some cases, programs have been modified to make it harder to exploit them using environment variables. Historically, many attacks exploited the way the command shell handled the `IFS` environment variable, but most of today's shells (including GNU bash) have been modified to make `IFS` harder to exploit.

What's the IFS problem?

Although it's not as serious a problem today, the `IFS` environment variable once caused many security problems in older Unix shells. `IFS` was used to determine what separated words in commands sent to the original Unix Bourne shell, and was passed down like any other environment variable. Normally the `IFS` variable would have the value of a space, a tab, and a newline -- any of those characters would be treated like a space character. But attackers could then set `IFS` to sneaky values, for example, they might add a "/" to `IFS`. Then, when the shell tried to run `/bin/ls`, the old shell would interpret "/" just like a space character -- meaning that the shell would run the "bin" program (wherever it could find one) with the "ls" option! The attacker would then provide a "bin" program that the program could find.

Thankfully, most of today's shells counter this by at least automatically resetting the `IFS` variable when they start -- and that includes GNU bash, the usual shell for GNU/Linux systems. GNU bash also limits the use of `IFS` so it's only used on the results of expansions. This means that `IFS` is used less often, and, thus, it's much less dangerous (the original sh split all words using `IFS`, even commands). Unfortunately, not all shells protect themselves (*Practical Unix & Internet Security* -- see Resources for a link -- has sample code to test this). And although this particular problem has been (for the most part) countered, it exemplifies the subtle problems that can occur from unchecked environment variables.

Unfortunately, while this hardening is a good idea, it's not enough -- you still need to deal with environment variables carefully. An extremely important (though complicated) example involves how all programs are run on Unix-like systems. Unix-like systems (including GNU/Linux) run programs by first running a system loader (it's /lib/ld-linux.so.2 on most GNU/Linux systems), which then locates and loads the necessary shared libraries. The loader is normally controlled by -- you guessed it -- environment variables.

On most Unix-like systems, the loader's search for libraries normally begins with any directories listed in the environment variable `LD_LIBRARY_PATH`. I should note that `LD_LIBRARY_PATH` works on many Unix-like systems, but not all; HP-UX uses the environment variable `SHLIB_PATH`, and AIX uses `LIBPATH` instead. Also, in GNU-based systems (including GNU/Linux), the list of libraries specified in the environment variable `LD_PRELOAD` is loaded first and overrides everything else.

The problem is that if an attacker can control the underlying libraries used by a program, the attacker can completely control the program. For example, imagine that the attacker could run /usr/bin/passwd (a privileged program that lets you change your password), but uses the environment variables to change the libraries used by the program. An attacker could write their own version of crypt(3), the password encryption function, and when the privileged program tries to call the library, the attacker can make the program do anything -- including allowing permanent, unlimited control over the system. Today's loaders counter this problem by detecting if the program is setuid/setgid, and if it is, they ignore environment variables such as `LD_PRELOAD` and `LD_LIBRARY_PATH`.

So, are we safe? No. If that malicious `LD_PRELOAD` or `LD_LIBRARY_PATH` value isn't erased by the setuid/setgid program, it will be passed down to other programs and cause the very problem the loader is trying to counter. Thus, the loader makes it *possible* to write secure programs, but you still have to protect against malicious environment variables. And that still doesn't deal with the problem of undocumented environment variables.

For secure setuid/setgid programs, the only safe thing to do is to always "extract and erase" environment variables at the beginning of the program:

- Extract the environment variables that you actually need (if any).
- Erase the entire environment. In C/C++, erasing the environment can be done by including `<unistd.h>` and then setting the `environ` variable to NULL (do this very early, in particular before creating any threads).
- Set just the environment variables you need to safe values. One environment value you'll almost certainly re-add is `PATH`, the list of directories to search for programs. Typically `PATH` should just be set to /bin:/usr/bin or some similar value. Don't include the current directory in `PATH`, which can be written as "." or even as a blank entry (so a colon at the beginning or end would probably be exploitable). Typically you'll also set `IFS` (to its default of " \t\n" -- space, tab, and newline) and `TZ` (timezone). Others you might set are `HOME` and `SHELL`. Your application might need a few more, but limit them -- don't accept data from a potential attacker unless it's critically needed.

Files

As mentioned in my previous installment, don't trust filenames that can be set by an attacker. Linux and Unix allow just about any series of characters to be a filename, so if you're

traversing a directory or accepting a filename from an attacker, be prepared. Attackers can create filenames with leading "-", filenames with special characters such as "&", and so on.

Don't trust file contents that can be controlled by untrusted users. That includes files viewed or edited by a program if they might be mailed by an attacker. For example, the popular text editor vim version 5.7, when asked to edit a file, would look for an embedded `statusline` command to set information on its status line, and that command could in turn execute an arbitrary shell program. An attacker could e-mail a specially rigged file to the victim, and if the victim used vim to read or edit it, the victim would run whatever program the attacker wanted. Oops.

Avoid getting configuration information from the current directory, because a user might view a directory controlled by an attacker who has created a malicious configuration file (for example, the attacker may have sent a compressed directory with the data and a malicious configuration file). Instead, get configuration information from /etc, the user's home directory, and/or the desktop environment's library for getting configuration information. It's a common convention to store configuration information and other information in "~/.*program-name*"; the period means it won't clutter normal displays. If you really must get configuration information from the current directory, aggressively check all data from it.

Don't let attackers control any temporary files. I suggest placing temporary directories inside a user's home directory if the user is trusted. If that isn't acceptable, use secure methods to create and use temporary files (I'll discuss how to securely create temporary files in a later article).

File descriptors

Sneaky attackers may start a program but do strange things to its standard input, standard output, or standard error. For example, an attacker might close one or more of them so that the next file you open is also where normal output goes. This is especially a problem for setuid/setgid programs. Some of today's Unix-like systems counter this, but not all.

One way a setuid/setgid program can counter this attack is to repeatedly open up /dev/null using `open()` until the file descriptor's value is more than 2 (you must do this before opening files, preferably early in the program initialization). Then, if the first call to `open()` returns 2 or less, exit without printing any messages. By first repeatedly opening up /dev/null, you protect yourself from yourself -- bad things won't happen if you accidentally try to open files and then print an error message. There's no need to print error messages for this case since file descriptors 0 through 2 are only closed if an attacker is trying to subvert your program.

Command line

Programs can be started up with data from the command line -- but can you trust that data? Setuid/setgid programs in particular cannot. If you can't trust the data, be prepared for anything -- large arguments, a huge number of arguments, improbable characters, and so on. Note that the name of the program is just argument number 0 in the command line values -- *don't* trust the program name, since an attacker can change it.

Also, try to design your command-line syntax so that it's easier to use securely. For example, support the standard "--" (double-dash) option that means "no more options," so that scripts

can use the option to foil attackers who create filenames (like "-fr") that begin with dash. Otherwise, an attacker can create "-fr" as a file and try to talk users into running "yourcommand *"; your program may then misinterpret the filename ("-fr") as an option.

Graphical user interface (GUI)

Here's a recipe for disaster: a process has special privileges (for instance, if it's setuid/setgid), it uses the operating system's graphical user interface (GUI) libraries, and the GUI user isn't totally trusted. The problem is that GUI libraries (including those on Unix, Linux, and Windows) simply aren't designed to be used that way. Nor would it make sense to try to do so -- GUI libraries are huge and depend on large substructures, so it would be difficult to fully analyze all that code for their security properties. The GTK+ GUI library even halts if it detects that it's running in a setuid program, because it's not supposed to be used that way (kudos to the GTK+ developers for proactively preventing this security problem).

Does that mean that you're doomed to the command line? No. Break your program into smaller parts, have an unprivileged part implement the GUI, and have a separate part implement the privileged operations. Here are some common ways to do this:

- It's often easiest to implement the privileged operations as a command-line program that's called by the GUI -- that way you get both a GUI and command-line interface (CLI) "for free," simplifying scripting and debugging. Typically the CLI privileged program is a setuid/setgid program. The privileged program, of course, must defend itself from all attacks, but this approach usually means that the part of the program that must be secured is much smaller and easier to defend.
- If you need high-speed communication, start up the program as a privileged program, split it into separate processes that can securely communicate, and then have one process permanently drop its privileges and run the GUI.
- Another approach is to implement a privileged server that responds to requests, and then create the GUI as a client.
- Use a Web interface; create a privileged server and use a Web browser as the client. This is really a special case of the previous method, but it's so flexible that it's often worth considering. You'll need to secure it just like any other Web application, which brings us to the problem of network data.

Network data

If data comes from a network, you should usually treat it as highly untrusted. Don't trust the "source IP" address, the HTTP "Referrer" header value, or similar data to tell you where the data really came from; those values come from the sender and can be forged. Be careful of values from the domain name system (DNS); DNS implements a distributed database, and some of those values may be supplied by the attacker.

If you have a client/server system, the server should never trust the client. The client data could be manipulated before reaching the server, the client program might have been modified, or attackers could have created their own client (many have!). If you're getting data from a Web browser, remember that Web cookies, HTML form data, URLs, and so on can be set by the user to arbitrary values. This is a common problem in Web shopping-cart applications; many of these applications use hidden HTML form fields to store product information (such as price) and related information (such as shipping costs), and blindly

accept these values when users send them. Not only can users set product prices to low values or zero, in some cases they can even set negative prices to receive the merchandise and an additional cash bonus. Remember that you have to check *all* data; some Web shopping carts check the product data but forget to check the shipping price.

If you're writing a Web application, limit `GET` requests to queries for data. Don't let `GET` requests actually change data (such as transferring money) or other activities. Users can be easily fooled into clicking on malicious hyperlinks in their Web browsers, which then send `GET` requests. Instead, if you get a `GET` request for some action other than a query, send back a confirmation message of the form "you asked me to do X, is that okay? (Ok, Cancel)" Note that limiting `GET` queries won't help you with the problem of incorrect client data (as discussed in the previous paragraph) -- servers still need to check data from their clients!

Miscellaneous

Programs have many other inputs, such as the current directory, signals, memory maps, System V IPC, the umask, and the state of the filesystem. Armed with the information you've gained here, the important thing is to not overlook these things as inputs, even though they don't always smell like inputs.

Back to top

Conclusions

Secure programs must check every untrusted input channel, and doing so can eliminate a lot of problems. But that's not enough. Sometimes, even just reading data can be a security vulnerability -- before the data is even checked! And processing the data can cause the program to fail in horrific ways. We're talking about the #1 security vulnerability today -- the buffer overflow. My next installment will discuss what this vulnerability is, how to counter it, and why there's hope that this will become less of a problem in the future.

Resources

- Read all of the installments in David's *Secure programmer* column. "Developing secure programs" covers terminology and other basics, and "Validating input discusses checking various data types.
- David's book *Secure Programming for Linux and Unix HOWTO* (Wheeler, March 2003) gives a detailed account of how to develop secure software. Chapter 7 discusses setuid and limiting privileges in detail.
- "SAP R/3 Web Application Server Demo for Linux: root exploit" by Jochen Hein (Bugtraq, 29 April 2001) discusses the SAP vulnerability. This vulnerability is Bugtraq id 2662 and CVE vulnerability CVE-2001-0366. Run `chmod u-s` as a workaround to disable it.
- "VIM statusline Text-Embedded Command Execution Vulnerability" (Bugtraq, 26 March 2001), Bugtraq id 2510, discusses the vim vulnerability. This vulnerability is

CVE vulnerability CVE-2001-0408 and was originally revealed as Red Hat RHSA-2001:008-04. You can disable it by turning off the `statusline` or `stl` option in .vimrc.

- "Multiple Vendor Web Shopping Cart Hidden Form Field Vulnerability" (Bugtraq, 1 Feb 2000), Bugtraq id 1237, discusses why blindly accepting product price values from users is a bad idea and the large number of Web applications with this error.

- "Well Known Flaw in Web Cart Software Remains Wide Open" by Beyond-Security's SecuriTeam.com notes that not only do some shopping carts fail to check product prices, some check product prices but not shipping prices (so negative shipping prices give an unjustified discount).

- David's article "Program Library HOWTO" (Wheeler, 11 April 2003) discusses how libraries (including shared libraries) are handled in GNU/Linux.

- "Why GTK_MODULES is not a security hole" by Owen Taylor (GTK.org, 2 January 2000) explains why "writing setuid and setgid programs using GTK+ is bad idea and will never be supported by the GTK+ team," and notes that current versions of GTK+ will not run setuid at all.

- Adam Shostack has posted a copy of the setuid(7) man page.

- A copy of the environ(5) man page is posted at Princeton's Computer Science department.

- Some nice discussion is included in the classic *Practical Unix & Internet Security, 3rd Edition* by Simson Garfinkel, Gene Spafford, and Alan Schwartz (O'Reilly & Associates, 2003). Chapter 11 discusses the `IFS` environment variable.

- Cameron Laird discusses server security in "Server clinic: Practical Linux security" on *developerWorks*.

- Gary McGraw and John Viega offer 10 points to keep in mind when building a secure system in the "Software security principles" series on *developerWorks*. In "Building secure software: Selecting technologies," Gary and John explore common choices faced by designers and programmers.

- Read more about Linux in the IBM developerWorks Linux section.

- IBM Research Security group has a number of security-related projects, including Internet security, Java security, cryptography, and data hiding.

- The IBM white paper "Secure Internet Applications on the AS/400 system" primarily discusses the Secure Sockets Layer (SSL) protocol.

- The AS/400 Security pages on Internet Security are a good starting point for learning more about network security.

About the author

David A. Wheeler is an expert in computer security and has long worked in improving development techniques for large and high-risk software systems. Mr. Wheeler is a validator for the Common Criteria. Mr. Wheeler also wrote the Springer-Verlag book *Ada95: The Lovelace Tutorial*, and is the co-author and lead editor of the IEEE book *Software Inspection: An Industry Best Practice*. This article presents the opinions of the author and does not necessarily represent the position of the Institute for Defense Analyses. You can contact David at dwheelerNOSPAM@dwheeler.com.