NAME

perlsec - Perl security

DESCRIPTION

Perl is designed to make it easy to program securely even when running with extra privileges, like setuid or setgid programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more builtin functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

SECURITY VULNERABILITY CONTACT INFORMATION

If you believe you have found a security vulnerability in Perl, please email perl5-security-report@perl.org with details. This points to a closed subscription, unarchived mailing list. Please only use this address for security issues in the Perl core, not for modules independently distributed on CPAN.

SECURITY MECHANISMS AND CONCERNS

Taint mode

Perl automatically enables a set of special security checks, called *taint mode*, when it detects its program running with differing real and effective user or group IDs. The setuid bit in Unix permissions is mode 04000, the setgid bit mode 02000; either or both may be set. You can also enable taint mode explicitly by using the **-T** command line flag. This flag is *strongly* suggested for server programs and any program run on behalf of someone else, such as a CGI script. Once taint mode is on, it's on for the remainder of your script.

While in this mode, Perl takes special precautions called *taint checks* to prevent both obvious and subtle traps. Some of these checks are reasonably simple, such as verifying that path directories aren't writable by others; careful programmers have always used checks like these. Other checks, however, are best supported by the language itself, and it is these checks especially that contribute to making a set-id Perl program more secure than the corresponding C program.

You may not use data derived from outside your program to affect something else outside your program--at least, not by accident. All command line arguments, environment variables, locale information (see *perllocale*), results of certain system calls (`readdir()`, `readlink()`, the variable of `shmread()`, the messages returned by `msgrcv()`, the password, gcos and shell fields returned by the `getpwxxx()` calls), and all file input are marked as "tainted". Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes, **with the following exceptions**:

- Arguments to `print` and `syswrite` are **not** checked for taintedness.
- Symbolic methods

  ```
  $obj->$method(@args);
  ```

  and symbolic sub references

  ```
  &{$foo}(@args);
  $foo->(@args);
  ```

  are not checked for taintedness. This requires extra carefulness unless you want external data to affect your control flow. Unless you carefully limit what these symbolic values are, people are able to call functions **outside** your Perl code, such as POSIX::system, in which case they are able to run arbitrary external code.
- Hash keys are **never** tainted.

For efficiency reasons, Perl takes a conservative view of whether data is tainted. If an expression contains tainted data, any subexpression may be considered tainted, even if the value of the

subexpression is not itself affected by the tainted data.

Because taintedness is associated with each scalar value, some elements of an array or hash can be tainted and others not. The keys of a hash are **never** tainted.

For example:

```
    $arg = shift;  # $arg is tainted
    $hid = $arg . 'bar'; # $hid is also tainted
    $line = <>;   # Tainted
    $line = <STDIN>;  # Also tainted
    open FOO, "/home/me/bar" or die $!;
    $line = <FOO>;  # Still tainted
    $path = $ENV{'PATH'}; # Tainted, but see below
    $data = 'abc';  # Not tainted

    system "echo $arg";  # Insecure
    system "/bin/echo", $arg; # Considered insecure
    # (Perl doesn't know about /bin/echo)
    system "echo $hid";  # Insecure
    system "echo $data"; # Insecure until PATH set

    $path = $ENV{'PATH'}; # $path now tainted

    $ENV{'PATH'} = '/bin:/usr/bin';
    delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

    $path = $ENV{'PATH'}; # $path now NOT tainted
    system "echo $data"; # Is secure now!

    open(FOO, "< $arg"); # OK - read-only file
    open(FOO, "> $arg");  # Not OK - trying to write

    open(FOO,"echo $arg|"); # Not OK
    open(FOO,"-|")
 or exec 'echo', $arg; # Also not OK

    $shout = `echo $arg`; # Insecure, $shout now tainted

    unlink $data, $arg;  # Insecure
    umask $arg;   # Insecure

    exec "echo $arg";  # Insecure
    exec "echo", $arg;  # Insecure
    exec "sh", '-c', $arg; # Very insecure!

    @files = <*.c>;  # insecure (uses readdir() or similar)
    @files = glob('*.c'); # insecure (uses readdir() or similar)

    # In either case, the results of glob are tainted, since the list of
    # filenames comes from outside of the program.

    $bad = ($arg, 23);  # $bad will be tainted
```

```
    $arg, `true`;  # Insecure (although it isn't really)
```

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure $ENV{PATH}".

The exception to the principle of "one tainted value taints the whole expression" is with the ternary conditional operator `?:`. Since code with a ternary conditional

```
    $result = $tainted_value ? "Untainted" : "Also untainted";
```

is effectively

```
    if ( $tainted_value ) {
        $result = "Untainted";
    } else {
        $result = "Also untainted";
    }
```

it doesn't make sense for `$result` to be tainted.

Laundering and Detecting Tainted Data

To test whether a variable contains tainted data, and whose use would thus trigger an "Insecure dependency" message, you can use the `tainted()` function of the Scalar::Util module, available in your nearby CPAN mirror, and included in Perl starting from the release 5.8.0. Or you may be able to use the following `is_tainted()` function.

```
    sub is_tainted {
        local $@;   # Don't pollute caller's value.
        return ! eval { eval("#" . substr(join("", @_), 0, 0)); 1 };
    }
```

This function makes use of the fact that the presence of tainted data anywhere within an expression renders the entire expression tainted. It would be inefficient for every operator to test every argument for taintedness. Instead, the slightly more efficient and conservative approach is used that if any tainted value has been accessed within the same expression, the whole expression is considered tainted.

But testing for taintedness gets you only so far. Sometimes you have just to clear your data's taintedness. Values may be untainted by using them as keys in a hash; otherwise the only way to bypass the tainting mechanism is by referencing subpatterns from a regular expression match. Perl presumes that if you reference a substring using $1, $2, etc. in a non-tainting pattern, that you knew what you were doing when you wrote that pattern. That means using a bit of thought--don't just blindly untaint anything, or you defeat the entire mechanism. It's better to verify that the variable has only good characters (for certain values of "good") rather than checking whether it has any bad characters. That's because it's far too easy to miss bad characters that you never thought of.

Here's a test to make sure that the data contains nothing but "word" characters (alphabetics, numerics, and underscores), a hyphen, an at sign, or a dot.

```
    if ($data =~ /^([-\@\w.]+)$/) {
 $data = $1;    # $data now untainted
    } else {
 die "Bad data in '$data'";  # log this somewhere
    }
```

This is fairly secure because `/\w+/` doesn't normally match shell metacharacters, nor are dot, dash, or at going to mean something special to the shell. Use of `/.+/` would have been insecure in theory

because it lets everything through, but Perl doesn't check for that. The lesson is that when untainting, you must be exceedingly careful with your patterns. Laundering data using regular expression is the *only* mechanism for untainting dirty data, unless you use the strategy detailed below to fork a child of lesser privilege.

The example does not untaint `$data` if `use locale` is in effect, because the characters matched by `\w` are determined by the locale. Perl considers that locale definitions are untrustworthy because they contain data from outside the program. If you are writing a locale-aware program, and want to launder data with a regular expression containing `\w`, put `no locale` ahead of the expression in the same block. See *"SECURITY" in perllocale* for further discussion and examples.

Switches On the "#!" Line

When you make a script executable, in order to make it usable as a command, the system will pass switches to perl from the script's #! line. Perl checks that any command line switches given to a setuid (or setgid) script actually match the ones set on the #! line. Some Unix and Unix-like environments impose a one-switch limit on the #! line, so you may need to use something like `-wU` instead of `-w -U` under such systems. (This issue should arise only in Unix or Unix-like environments that support #! and setuid or setgid scripts.)

Taint mode and @INC

When the taint mode (`-T`) is in effect, the "." directory is removed from `@INC`, and the environment variables `PERL5LIB` and `PERLLIB` are ignored by Perl. You can still adjust `@INC` from outside the program by using the `-I` command line option as explained in *perlrun*. The two environment variables are ignored because they are obscured, and a user running a program could be unaware that they are set, whereas the `-I` option is clearly visible and therefore permitted.

Another way to modify `@INC` without modifying the program, is to use the `lib` pragma, e.g.:

```
perl -Mlib=/foo program
```

The benefit of using `-Mlib=/foo` over `-I/foo`, is that the former will automagically remove any duplicated directories, while the later will not.

Note that if a tainted string is added to `@INC`, the following problem will be reported:

```
Insecure dependency in require while running with -T switch
```

Cleaning Up Your Path

For "Insecure `$ENV{PATH}`" messages, you need to set `$ENV{'PATH'}` to a known value, and each directory in the path must be absolute and non-writable by others than its owner and group. You may be surprised to get this message even if the pathname to your executable is fully qualified. This is *not* generated because you didn't supply a full path to the program; instead, it's generated because you never set your PATH environment variable, or you didn't set it to something that was safe. Because Perl can't guarantee that the executable in question isn't itself going to turn around and execute some other program that is dependent on your PATH, it makes sure you set the PATH.

The PATH isn't the only environment variable which can cause problems. Because some shells may use the variables IFS, CDPATH, ENV, and BASH_ENV, Perl checks that those are either empty or untainted when starting subprocesses. You may wish to add something like this to your setid and taint-checking scripts.

```
    delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Make %ENV safer
```

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens and such **after** properly dropping any special user (or group!) privileges. Perl doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The

tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

Perl does not call the shell to expand wild cards when you pass `system` and `exec` explicit parameter lists instead of strings with possible shell wildcards in them. Unfortunately, the `open`, `glob`, and backtick functions provide no such alternate calling convention, so more subterfuge will be required.

Perl provides a reasonably safe way to open a file or pipe from a setuid or setgid program: just create a child process with reduced privilege who does the dirty work for you. First, fork a child using the special `open` syntax that connects the parent and child by a pipe. Now the child resets its ID set and any other per-process attributes, like environment variables, umasks, current working directories, back to the originals or known safe values. Then the child process, which no longer has any special permissions, does the `open` or other system call. Finally, the child passes the data it managed to access back to the parent. Because the file or pipe was opened in the child while running under less privilege than the parent, it's not apt to be tricked into doing something it shouldn't.

Here's a way to do backticks reasonably safely. Notice how the `exec` is not called with a string that the shell could expand. This is by far the best way to call something that might be subjected to shell escapes: just never call the shell at all.

```
    use English;
    die "Can't fork: $!" unless defined($pid = open(KID, "-|"));
    if ($pid) {           # parent
        while (<KID>) {
            # do something
        }
        close KID;
    } else {
        my @temp     = ($EUID, $EGID);
        my $orig_uid = $UID;
        my $orig_gid = $GID;
        $EUID = $UID;
        $EGID = $GID;
        # Drop privileges
        $UID  = $orig_uid;
        $GID  = $orig_gid;
        # Make sure privs are really gone
        ($EUID, $EGID) = @temp;
        die "Can't drop privileges"
            unless $UID == $EUID  && $GID eq $EGID;
        $ENV{PATH} = "/bin:/usr/bin"; # Minimal PATH.
# Consider sanitizing the environment even more.
        exec 'myprog', 'arg1', 'arg2'
            or die "can't exec myprog: $!";
    }
```

A similar strategy would work for wildcard expansion via `glob`, although you can use `readdir` instead.

Taint checking is most useful when although you trust yourself not to have written a program to give away the farm, you don't necessarily trust those who end up using it not to try to trick it into doing something bad. This is the kind of security checking that's useful for set-id programs and programs launched on someone else's behalf, like CGI programs.

This is quite different, however, from not even trusting the writer of the code not to try to do something evil. That's the kind of trust needed when someone hands you a program you've never seen before and says, "Here, run this." For that kind of safety, you might want to check out the Safe module, included standard in the Perl distribution. This module allows the programmer to set up special compartments in which all system operations are trapped and namespace access is carefully

controlled. Safe should not be considered bullet-proof, though: it will not prevent the foreign code to set up infinite loops, allocate gigabytes of memory, or even abusing perl bugs to make the host interpreter crash or behave in unpredictable ways. In any case it's better avoided completely if you're really concerned about security.

Security Bugs

Beyond the obvious problems that stem from giving special privileges to systems as flexible as scripts, on many versions of Unix, set-id scripts are inherently insecure right from the start. The problem is a race condition in the kernel. Between the time the kernel opens the file to see which interpreter to run and when the (now-set-id) interpreter turns around and reopens the file to interpret it, the file in question may have changed, especially if you have symbolic links on your system.

Fortunately, sometimes this kernel "feature" can be disabled. Unfortunately, there are two ways to disable it. The system can simply outlaw scripts with any set-id bit set, which doesn't help much. Alternately, it can simply ignore the set-id bits on scripts.

However, if the kernel set-id script feature isn't disabled, Perl will complain loudly that your set-id script is insecure. You'll need to either disable the kernel set-id script feature, or put a C wrapper around the script. A C wrapper is just a compiled program that does nothing except call your Perl program. Compiled programs are not subject to the kernel bug that plagues set-id scripts. Here's a simple wrapper, written in C:

```
    #define REAL_PATH "/path/to/script"
    main(ac, av)
char **av;
    {
execv(REAL_PATH, av);
    }
```

Compile this wrapper into a binary executable and then make *it* rather than your script setuid or setgid.

In recent years, vendors have begun to supply systems free of this inherent security bug. On such systems, when the kernel passes the name of the set-id script to open to the interpreter, rather than using a pathname subject to meddling, it instead passes */dev/fd/3*. This is a special file already opened on the script, so that there can be no race condition for evil scripts to exploit. On these systems, Perl should be compiled with `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. The *Configure* program that builds Perl tries to figure this out for itself, so you should never have to specify this yourself. Most modern releases of SysVr4 and BSD 4.4 use this approach to avoid the kernel race condition.

Protecting Your Programs

There are a number of ways to hide the source to your Perl programs, with varying levels of "security".

First of all, however, you *can't* take away read permission, because the source code has to be readable in order to be compiled and interpreted. (That doesn't mean that a CGI script's source is readable by people on the web, though.) So you have to leave the permissions at the socially friendly 0755 level. This lets people on your local system only see your source.

Some people mistakenly regard this as a security problem. If your program does insecure things, and relies on people not knowing how to exploit those insecurities, it is not secure. It is often possible for someone to determine the insecure things and exploit them without viewing the source. Security through obscurity, the name for hiding your bugs instead of fixing them, is little security indeed.

You can try using encryption via source filters (Filter::* from CPAN, or Filter::Util::Call and Filter::Simple since Perl 5.8). But crackers might be able to decrypt it. You can try using the byte code compiler and interpreter described below, but crackers might be able to de-compile it. You can try

using the native-code compiler described below, but crackers might be able to disassemble it. These pose varying degrees of difficulty to people wanting to get at your code, but none can definitively conceal it (this is true of every language, not just Perl).

If you're concerned about people profiting from your code, then the bottom line is that nothing but a restrictive license will give you legal security. License your software and pepper it with threatening statements like "This is unpublished proprietary software of XYZ Corp. Your access to it does not give you permission to use it blah blah blah." You should see a lawyer to be sure your license's wording will stand up in court.

Unicode

Unicode is a new and complex technology and one may easily overlook certain security pitfalls. See *perluniintro* for an overview and *perlunicode* for details, and *"Security Implications of Unicode" in perlunicode* for security implications in particular.

Algorithmic Complexity Attacks

Certain internal algorithms used in the implementation of Perl can be attacked by choosing the input carefully to consume large amounts of either time or space or both. This can lead into the so-called *Denial of Service* (DoS) attacks.

- Hash Algorithm - Hash algorithms like the one used in Perl are well known to be vulnerable to collision attacks on their hash function. Such attacks involve constructing a set of keys which collide into the same bucket producing inefficient behavior. Such attacks often depend on discovering the seed of the hash function used to map the keys to buckets. That seed is then used to brute-force a key set which can be used to mount a denial of service attack. In Perl 5.8.1 changes were introduced to harden Perl to such attacks, and then later in Perl 5.18.0 these features were enhanced and additional protections added.

  At the time of this writing, Perl 5.18.0 is considered to be well-hardened against algorithmic complexity attacks on its hash implementation. This is largely owed to the following measures mitigate attacks:

  Hash Seed Randomization

  In order to make it impossible to know what seed to generate an attack key set for, this seed is randomly initialized at process start. This may be overridden by using the PERL_HASH_SEED environment variable, see *"PERL_HASH_SEED" in perlrun*. This environment variable controls how items are actually stored, not how they are presented via `keys`, `values` and `each`.

  Hash Traversal Randomization

  Independent of which seed is used in the hash function, `keys`, `values`, and `each` return items in a per-hash randomized order. Modifying a hash by insertion will change the iteration order of that hash. This behavior can be overridden by using `hash_traversal_mask()` from *Hash::Util* or by using the PERL_PERTURB_KEYS environment variable, see *"PERL_PERTURB_KEYS" in perlrun*. Note that this feature controls the "visible" order of the keys, and not the actual order they are stored in.

  Bucket Order Perturbance

  When items collide into a given hash bucket the order they are stored in the chain is no longer predictable in Perl 5.18. This has the intention to make it harder to observe a collisions. This behavior can be overridden by using the PERL_PERTURB_KEYS environment variable, see *"PERL_PERTURB_KEYS" in perlrun*.

  New Default Hash Function

  The default hash function has been modified with the intention of making it harder to infer the hash seed.

  Alternative Hash Functions

The source code includes multiple hash algorithms to choose from. While we believe that the default perl hash is robust to attack, we have included the hash function Siphash as a fall-back option. At the time of release of Perl 5.18.0 Siphash is believed to be of cryptographic strength. This is not the default as it is much slower than the default hash.

Without compiling a special Perl, there is no way to get the exact same behavior of any versions prior to Perl 5.18.0. The closest one can get is by setting PERL_PERTURB_KEYS to 0 and setting the PERL_HASH_SEED to a known value. We do not advise those settings for production use due to the above security considerations.

Perl has never guaranteed any ordering of the hash keys, and the ordering has already changed several times during the lifetime of Perl 5. Also, the ordering of hash keys has always been, and continues to be, affected by the insertion order and the history of changes made to the hash over its lifetime.

Also note that while the order of the hash elements might be randomized, this "pseudo-ordering" should **not** be used for applications like shuffling a list randomly (use `List::Util::shuffle()` for that, see *List::Util*, a standard core module since Perl 5.8.0; or the CPAN module `Algorithm::Numerical::Shuffle`), or for generating permutations (use e.g. the CPAN modules `Algorithm::Permute` or `Algorithm::FastPermute`), or for any cryptographic applications.

Tied hashes may have their own ordering and algorithmic complexity attacks.

- Regular expressions - Perl's regular expression engine is so called NFA (Non-deterministic Finite Automaton), which among other things means that it can rather easily consume large amounts of both time and space if the regular expression may match in several ways. Careful crafting of the regular expressions can help but quite often there really isn't much one can do (the book "Mastering Regular Expressions" is required reading, see *perlfaq2*). Running out of space manifests itself by Perl running out of memory.

- Sorting - the quicksort algorithm used in Perls before 5.8.0 to implement the sort() function is very easy to trick into misbehaving so that it consumes a lot of time. Starting from Perl 5.8.0 a different sorting algorithm, mergesort, is used by default. Mergesort cannot misbehave on any input.

See *http://www.cs.rice.edu/~scrosby/hash/* for more information, and any computer science textbook on algorithmic complexity.

SEE ALSO

perlrun for its description of cleaning up environment variables.