Analysis of Perl's Taint Mode

Andrew Hurst <abhurst@ucdavis.edu>

June 8, 2004

Abstract

A program that takes no input and produces no output is not a very useful program. Unfortunately, as soon as a program starts taking input from users a potential security hole is opened where specially crafted data could be given by the user. If the programmer did not think to check for malicious input, security violations could result. This paper will examine the approach that Perl provides to ensure that the data given by users of Perl programs is moderately secure, and how the requirements of the program affect what secure means and how it is tested for.

1 Overview

In the security climate of today, programmers can no longer assume that people will follow the rules just because they are there. This holds doubly true in web-enabled applications, where the feeling of anonymity encourages some people to try to find and exploit software weaknesses. Users will throw all sorts of data at a program hoping to cause it to crash or do something it wasn't supposed to.

One example might be a web-based email tool that takes the name of a mailbox as input. If the programmer assumed that the user would only enter valid mailboxes, then the program might not handle the case where a user entered `/etc/passwd` as the name of their mailbox. Thus the web-based email program now becomes a web-based file browser for the server.

To catch these types of errors, programmers have to be rigorous in the checking of any and all input coming from users. Of course, there are many more sources of input to programs than just user input; environment variables, command line options, and input from a configuration file all could contain malicious data.

The popular programming language Perl has provided a runtime option for helping to thwart these types of attacks by providing *taint mode*. When perl scripts are run with the `-T` flag, input derived from outside of the perl program will be flagged as "tainted", and will have to be "untainted" before it can be used as input to another program or system call. The authors of [9] sum it up nicely

```
#!/usr/bin/perl
# program1.pl
# given a filename, email it to myself. "Unsafe" version.
my $file = $ARGV[0];
`mail -s "Re: Your File" andrew\@hurstdog.org < $file`;
exit 0;
```

Figure 1: An example of using tainted data

with the statement "you may not use data derived from outside your program to affect something else outside your program".

It should be clarified here that this is a tool intended to help programmers protect their programs from users. Taint mode will not help a programmer protect against other possibly hostile programmers that will use their code. It is much too easy for other programmers to just falsely untaint the data. This is explained more in Section 3.5.

A simple example might help to clarify the intent and use of taint mode. Figure 1 contains a short perl program that reads in a file and emails it to the author at `andrew@hurstdog.org`.

Now before we can determine if this program is secure, we need to define what secure means. Assume that for this program the requirements are that it can only email files in the current directory or below, and that it should email the file to only myself with no side effects (it doesn't cc it to someone else, say, email a different file, or change the filesystem).

With these assumptions, there are two basic vulnerabilities in the program as it is written: (1) the filename given on the command line is accepted without question, and (2) the environment's `PATH` value is trusted. Because the filename is not checked, a file that the programmer did not intend to be emailed could be sent, for instance `/etc/passwd` or another file not in the users current directory or below. Trusting the `PATH` environment variable means that an attacker could modify the `PATH` to contain a directory of their choice early in the search order, and replace the `mail` program in that directory with one of their own. Thus they could get the author to email files that were meant to be emailed, execute code that was not meant to be executed, or any other of a number of dirty tricks that could be thought up.

Both of these vulnerabilities will be caught by running the script under taint mode, but they won't be fixed. When the `-T` flag is added to the first line of the script and re-run, the program execution is halted and the error `Insecure $ENV{PATH} while running with -T switch at program1.pl line 4` is displayed. Perl is complaining that the `PATH` environment variable is being relied upon by the script, yet it was supplied from outside of the program. This can be fixed by setting the path to a constant value.

```
#!/usr/bin/perl -T
# program2.pl
# given a filename, email it to myself.  "Safe" version.
my $file = $ARGV[0];
# get rid of any '..' that might be in the filename
$file =~ s|\.\.||g;
# only use the non-absolute version of the filename.
$file =~ m|^/*(.*?)$|;   # line 8
my $safe_file = $1;
# no need for the PATH to be trusted, Set it to a known value.
$ENV{PATH} = '/usr/bin';
# send myself the email
`mail -s "Re: Your File" andrew\@hurstdog.org < $safe_file`;
exit 0;
```

Figure 2: A more secure version of the program in Figure 1

After fixing that error, when run again the execution still halts but with the error `Insecure dependency in `` while running with -T switch at ./program1.pl line 5.` This is due to the `$file` variable being tainted, by virtue of it gaining its value from `$ARGV[0]`, which is tainted because it was supplied by the user. Just having `$file` tainted was not enough to trigger this error, however. Because it was used in the backtick operators (``), it was fed to a shell on the system. This is a big vulnerability. To fix this I untainted `$file` by doing some checks with perl's regular expression support, which allowed me to run the program successfully under taint mode. A more secure version of the program in Figure 1 is shown in Figure 2.

Thus by running the program under taint mode, we were able to determine unsafe practices and places that users could inject malicious data into the program. It is important to notice in the script above that I had to untaint the data myself. Thus if there were any bugs in my untainting, or I didn't untaint it effectively enough, the program could still be insecure. In fact, the "secure" program in Figure 2 is still insecure given the earlier requirements.

If the user supplies a filename of `\/etc/passwd` it will pass the tests above and email the file `/etc/passwd`. For more on this see Sections 3.5 and 4.2.1.

2 Related Work

Much work has been done in this area, from other languages that implement similar mechanisms as perl (Section 2.1) to program

analyzer's that run separately from the regular development cycle to look for vulnerabilities (Section 2.2).

2.1 Ruby

Along with Perl, the Ruby programming language has a taint checking mechanism built into it [8]. Ruby's implementation works by having multiple Safe Levels that a program can run under. At each of the five safety levels different amounts of security checks are performed; from no checks (Safe Level 0) to essentially requiring that all loaded code runs in a sand box (Safe Level 4 — similar to using the `Safe` module in Perl's `CPAN`).

Ruby's Safe Levels provide a much finer-grained control over the running of scripts than Perl's taint mode. Safe level 1 in Ruby is equivalent to running with the `-T` flag in perl. The higher safety levels add checks that can disallow operations like loading libraries from world-writable locations, control whether newly created objects are tainted, and change all untainted objects to be read-only [8].

The other commonly used scripting languages Python and PHP do not include taint-like checks.

2.2 Format String Vulnerabilities

Perl's taint checks are similar to much of the format string vulnerability work that has gone on in recent years. For a little background, format string vulnerabilities are exploited by insecure calls to the C programming language's `printf()` and related functions [7]. These insecure calls would be structured such that some of the user's input to the program is used directly in the format string given to `printf()`. These vulnerabilities are not limited to the C language in particular, rather they are possible in any programming language that includes a `printf()`-like function and has programmers that make mistakes occasionally.

```
printf(buf);
```

Figure 3: A possibly insecure call to `printf()`

Figure 3 contains an example of this. If `buf` contains format specifiers like `%s` it will attempt to read these arguments off of the stack. If the user (as opposed to the programmer) is the one who supplied the data in `buf`, then most likely any format specifiers that are in the string will not have associated variables on the stack to print out. Thus the program will crash or will not print out the data that was intended.

An attacker that knows this can craft a format string specifically to provide data about the system that the programmer didn't intend to share, or get the program to run code for the attacker.

The authors of `cqual` solved this problem in C by using a type-based approach that is similar in spirit to taint mode in perl [7]. Their approach extends the type system of C at compile time (note that this isn't the normal developmental compile-time, but a separate compilation run to find er-

rors using `cqual`) to include the `tainted` and `untainted` type qualifiers. Because most unmodified C programs do not include these type qualifiers, the authors include with their system a set of annotated library functions for C that make use of the `tainted` and `untainted` type qualifiers. E.g. to note that `getchar()` returns a `tainted int`.

To determine if any function calls within the program are insecure, the system processes a generated Abstract Syntax Tree of the program to generate a database of type constraints which are solved to look for any inconsistencies.

For instance, assume a programmer calls `printf(buf)` in her program, and the set of annotated library functions notes that the first argument to printf should be an `untainted char *`. When `cqual` gets the generated abstract syntax tree, it will notice the call to `printf`, and the requirement for untainted input. `cqual` will then search the type constraint database to make sure that the variable `buf` was not derived from tainted input.

Intuitively, this approach produces no overly difficult problems. But when the type system of C is added to with type casts, pointers, and `void *` is taken into account, the problem becomes much more difficult. This highlights one benefit of perl's taint mode: because taint mode tracks taintedness through a flag on each expression, it doesn't care what the expression does, just whether or not the expression contains a tainted value or not. This will be examined more in Section 3.2.

Another approach to find security vulnerabilities in programs was used in Splint by adding annotations (essentially formatted notes within the comments of the program that include information about the variables and functions within it) [6]. Splint analyzes a program statically to determine possible security vulnerabilities through unsafe coding practices and annotations provided within the source. For instance if a comment `/*@untainted@*/` occurs before an argument in a function declaration, it means that that function requires untainted data. If during Splint's analysis of the program it notices that it is possible that tainted data can be passed to that function in the annotated argument, it will raise an error.

Splint does much more than just analyze for string format vulnerabilities and taintedness, however. It includes options for checking buffer overflows and adding what are essentially programming-by-contract restrictions to C programs.

Because Splint works by statically analyzing the program, there are limitations to what it can do due to undecidable problems. Taint mode and `cqual` get around this by analyzing at runtime, and using type checks (which don't include as many undecidable problems as static analysis) respectively.

3 Details

3.1 The Intent

Before we get into the details of taint mode, it would behoove us to give it some context by going into the intent of this feature. Taint

mode was not designed to be the be-all end-all of perl security. Its goal in not to catch every possible error that a program can make, but only to catch the stupid ones.

As the authors of the `perlsec` documentation so eloquently put it:

> The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.[4]

Think of taint mode as an airbag for your perl programs. Just because your car has an airbag does not mean that you don't pay attention when you drive. Nor does it mean that you don't care if you get into an accident or not. But if there is an accident it is nice that the airbag is there to do its part.

That is Perl's taint mode, a tool to help you write secure programs, but not the only help you'll ever need.

Related to this, taint mode will not protect you against malicious library writers or other programmers that use your program. As described in section 3.5 it is much too easy to work around if you have access to run arbitrary code to rely on this to protect your program from other programmers.

3.2 Determining Taintedness

As alluded to above, Perl's taint checking is a runtime check. As the program is running, each expression carries with it a flag that marks whether it contains tainted data or not. The taintedness of an expression is given by the taintedness of the variables and sub-expressions that make it up. However, by design, the Perl interpreter is not smart about determining taintedness.

```
# the rhs reduces to 2*1
# no matter what $t is.
$num = 2 *
    ( 2 - 1 /
        ( (0 * $t) + 1 )
    );
```

Figure 4: An example of over-zealous tainting

Rather than parsing each expression to see if all of the values in it affect the final value of it, if any of the variables in the expression are tainted the whole expression is marked as tainted [4]. This is done for implementation ease. In security related concerns it is almost always better to have a few minor false-positives than let any false-negatives get through.

For instance, in Figure 4 if `$t` contains tainted data then the entire right-hand-side of the expression will be marked tainted — and thus `$num` will be as well — irrespective of the fact that the value of `$num` is not dependent on the value of `$t`.

But how do we determine if `$t` is tainted in the first place? This is done by tracking the source of the data. As was mentioned earlier, any data from outside the program is automatically treated as tainted. This includes data from files, command line arguments, environment variables, locale information, and the return values of some system calls (readdir() and readlink(), among others) [4].

In Figure 1 the variable `$file` is tainted because its value is derived from `@ARGV`, the

```
#!/usr/bin/perl -T
# Perl only taint checks lines
# that get executed
if( 0 ) {
  $to   = 'andrew@hurstdog.org';
  $what = $ARGV[0];
  `mail -s "busted" $to < $what`;
}
else {
  print "OK\n";
}
```

Figure 5: A program that taint checks correctly, even though it contains possibly insecure code.

array of arguments to the perl program.

3.3 When Taint Checks Occur

Few perl programs would ever run successfully under taint mode if anything possibly tainted in the program caused it to stop running irrespective of whether the unsafe code got executed or not. Which is probably just as well, because to implement that type of checking the Perl interpreter would probably never finish analyzing any non-trivial program for control-flow paths to see if those statements could be executed!

Thus Perl only checks to see if a specific expression or variable contains tainted data if the program attempts to use it in an unsafe matter. Figure 5 contains a program very similar to `program1.pl` in Figure 1, but the insecure lines are set within an unreachable part of the program. When run, the program in Figure 5 runs correctly and outputs the string "OK" with no taint errors stopping execution.

The unsafe code is never run, and thus the program is never halted with errors.

3.4 Tainted Arguments

Any command that modifies files, directories, or processes, or invokes a sub-shell will fail with a taint check error if that command was issued with tainted data. This is to ensure that programs will not have unintended affects on the system in which they run. As an exception to this, the `print` and `syswrite` commands (and related, like `printf`) are not checked for tainted arguments.

To understand why the `print` and `syswrite` commands are not part of the group of functions that require untainted arguments, you have to know how localization works in perl.

When localization is turned on using the `locale` pragma, extra processing of strings happens throughout the program. This affects string sorting, comparing, and converting to and from numbers. Not all languages use the same alphabet as English, and not all sort the same. As well, some languages use '.' as a decimal point, others use ','. Thus when perl is converting from a number to a string, it needs to figure out how to write the decimal point.

This is done by reading figuring out the value of `LC_NUMERIC` for the current locale. `LC_NUMERIC` is one of a number of locale categories read from locale configuration files based on the locale the program is running

under. Because these values are read from external configuration files, their values are tainted [3].

The process of printing out some text when taking locale into account works as follows:

1. A command like `print("PI is almost exactly " . 22/7 );` is issued. Note that this line contains purely constant (untainted) data at this point.

2. Perl computes the value of `22/7` and converts it to a string using the value of `LC_NUMERIC` to determine the decimal point character. The string value of `22/7` is now tainted, as it used input from an external source to determine its value.

3. The string value of `22/7` is appended to the original string, and fed as an argument to `print()`. Note that the full string is tainted now.

4. `print()` prints the value of the string to the screen.

Were perl to ensure that no tainted values could be given as arguments to `print()`, when a user was running under `use locale;` they would never be able to print anything. Anything they tried to print would become tainted after they could do anything to prevent it, and before it was passed the `print()` function. Thus no print commands would succeed. Note that other variables than just `LC_NUMERIC` are used when `locale` is in effect, so this applies to more than just string arguments that contain numeric computations.

As an example of a function that requires untainted arguments, use `mkdir()`. This function works just like the shell command `mkdir` and creates a new directory with the name of the argument given to it. Were this function to not require untainted arguments, users of programs that made use of `mkdir()` could possibly create arbitrary directories. This would be even more of a problem if the program was running setuid as another user.

3.5 Untainting

If there were no way to untaint variables, then programs that run under taint mode would be rather useless. They couldn't run any outside commands, read configurations from a file and act on them, or do a number of useful things. For this reason there is a method to untaint variables.

To untaint a variable, you must use Perl's regular expression support with capturing. Line 8 in Figure 2 contains a match against the user-supplied filename. Within this match filenames that are considered valid are captured between the parenthesis.

On line 9 of the same program the variable `$safe_file` gets the value of the regular expression capture.

By requiring this method to untaint variables, perl is encouraging you to check the input to your program. Referring back to the intent of taint mode (Section 3.1), this method is not foolproof, however.

Figure 6 contains another version of the program in Figure 2. This program is definitely insecure given the requirements for the

```
#!/usr/bin/perl -T
# program3.pl
# given a filename, email it to myself. "Falsely Safe" version.
my $file = $ARGV[0];
$file =~ /(.*)/;
my $safe_file = $1;
$ENV{PATH} =~ /(.*)/;
$ENV{PATH} = $1;
`mail -s "Re: Your File" andrew\@hurstdog.org < $safe_file`;
exit 0;
```

Figure 6: An example of working around taint mode

emailing program. By using regular expression that capture the full input of whatever is given to the program, the programmer is in essence laundering the data to untaint it.

Because no checks are done on the data, the program is no more secure than if it were run without the `-T` flag.

3.6 Forced Taint Mode

When perl detects that it is running with differing real and effective user or group id's (i.e. setuid or setgid scripts), it forces running under taint mode. This is done to add an extra level of security for programs that need it the most.

To ensure that this feature is used, there is no way to turn off taint mode once it has been enabled for a program. Were there a way to do this, malicious programmers could just turn off taint mode first thing, then get the program to run setuid or setgid.

4 Evaluation

4.1 Benefits

4.1.1 Flexibility

Because taint mode does not decree what makes secure data or not — it merely flags possibly insecure data — it is very flexible. One person may use it to make sure that they check all given email addresses are valid, and another may use it to ensure that they run with a secure `PATH` environment variable.

By relying on the user to do the checks, any type of data that a perl program can read in can be taint checked, and later untainted.

This is also taint mode's greatest weakness, however. This is detailed more in Section 4.2.1

4.1.2 Ease of Implementation

Section 3.2 explains the simple algorithm that perl uses to propagate taintedness between expressions. Because of the simplic-

ity of this algorithm, the implementation of taint mode in perl is very straightforward. No difficult algorithms to determine type propagation, and no need to figure out how that taintedness with propagate.

A flag to mark if a given expression is tainted when that expression is generated, and a check to see if that flag is set on potentially unsafe system calls is all that is needed. Bugs due to complexity will be virtually nonexistent in the implementation. The major source of bugs for taint mode will be bugs of omission, described in Section 4.2.2.

4.2 Detriments

4.2.1 Only as Good as the Programmer

As mentioned in Section 4.1.1, each programmer is free to determine how to untaint and check the input that is provided to their program. Programmers come in varying degrees of skill, however, and one may write much stronger checks for the same input than another.

This points again to the fact that taint checking is not a panacea, but merely an antibiotic of sorts.

4.2.2 Implementation Bugs

Almost no program is without bugs, and perl is no exception. Taint mode has been known to have bugs of omission [2]. Possible sources of tainted data have been ignored in previous implementations of perl, and system calls that should require untainted data have not checked their arguments as well.

It is reasonable to assume that there probably still are bugs in perl's implementation of taint mode.

5 Similarity to the Biba Integrity Model

Comparisons of taint mode to formal security and integrity models are useful because it allows us to leverage the work done on the models to apply to taint mode.

By looking at the properties of the Biba and Bell-LaPadula Models, its easy to see that taint mode applies more to the Biba model, than to the Bell-LaPadula. This is because Bell-LaPadula is more concerned with security and access to data, and Biba is more concerned with the integrity of data. Taint mode does not restrict access to data based on if you have permissions, but only if the data is trustworthy or not. Thus the Biba model is a better fit.

5.1 Integrity Levels

To model taint mode in the Biba system, we create two integrity levels: Tainted and Untainted. Untainted is the more trustworthy of the two, so we make it the higher level. Untainted $<$ Tainted (read: Untainted dominates Tainted) in this model, as shown in Table 1.

The obvious case of classification is variables that are either tainted or untainted. Any variable that contains tainted data would be at the Tainted integrity level, and

Untainted (U)	*mkdir(), chown(), ` `, untainted variables, ...*
⏐	
Tainted (T)	*%ENV, @ARGV, tainted variables, ...*

Table 1: A Biba Integrity Model integrity graph for taint mode.

any variable that contains only untainted data would be at the Untainted integrity level.

To handle the case where taint check failures can occur when calling a function in an unsafe manner (i.e. an attempt to call `system()` with a tainted parameter) we classify all functions that require untainted data at the Untainted level.

Functions that return tainted values (and therefore the return values of those functions) are classified at the Tainted level.

Some functions however are merely carriers of tainted or untainted values. Examples of these are `+`, `-`, `.`, `map()`, and user-defined functions, among others. These functions act as the sum of the expressions that make them up, and whether or not they return tainted or untainted data depends on the data fed into them.

For example the return value of an identity function `id()` that just returns the value passed into it, would be classified at the integrity level of its arguments.

5.2 Weak Tranquility

But before we can fully model taint mode in the Biba model, we need to take into account tranquility. Strong Tranquility as it applies to the Biba model states that integrity levels may not change throughout the life of the system [5]. As tainted variables without the ability to untaint them would make taint mode rather useless, we cannot apply it here.

Rather, we will apply Weak Tranquility, which notes that security levels in a system may change as long as they do not change in a way that violates the security policies of the system.

This issue arises because simple reading and writing does not fully capture the semantics of expressions in perl, due to the restrictions on read in the low-water-mark policy.

Assume perl's assignment operator (`=`) was treated as a *read* between two variables, then once a variable was tainted it would never be able to become untainted. Take the example of an assignment between the subject `$s` and the object `$o`, `$s = $o`.

The integrity level of `$s` is given by the minimum of the integrity levels of `$s` and `$o` [5]. If `$s` is at the Tainted level and `$o` is at the Untainted level, `$s` would end up at the Tainted level after the assignment. This does not conform to the semantics of perl, which would leave `$s` untainted after this assignment.

To handle this case, I've used an implementation of weak tranquility to go along with *read* and *write*: *assign*. This operation fits into the model as follows (hereafter referred to as the *assignment rule*):

> Given $o_1, o_2 \in O$. If o_1 *assigns* o_2, then $i'(o_1) = i(o_2)$ and $o_1 = o_2$, where $i'(o_1)$ is the object's integrity level after the assignment.

Using the assignment rule, we can now successfully untaint data by using the capture operators as described near the end of the next section.

5.3 Following The Rules

For the purposes of this application of the Biba model, we consider *read* to be when a variable (or the return value of a function) is used in an expression; the subject doing the reading would be the expression the variable appears in, excluding anything to the left of the assignment operator. (See Table 2 for some examples).

Writes would occur when a variable is passed as an argument to a function; the object getting written to would be the function, the subject doing the writing would be the variable.

Assign is used only for the assignment operator, and works as described above.

The terms subject and object in the Biba Model rules will be used interchangeably with the term "expression". This corresponds to the expressions used internally in the parse tree of the perl interpreter, and thus the expressions that contain a tainted flag to track if they contain tainted data or not.

Using the low-water-mark policy, we can show that when a tainted variable is used in an expression (i.e. the variable is *read*), the resulting expression will be tainted as well.

The specific rule of the low-water-mark policy that provides this is rule 2, which states: If $s \in S$ reads $o \in O$, then $i'(s) = \min(i(s), i(o))$, where $i'(s)$ is the subject's integrity level after the read [5].

For example, look to Example 4 in Table 2. The security level of the full expression after the expression is executed will be $\min(i(\texttt{\$b}), i(\texttt{\$c}))$. Because of there only being two integrity levels, this is trivial to solve. If either $i(\texttt{\$b})$ or $i(\texttt{\$c})$ is at the Tainted level, then $i'(\texttt{\$b+\$c})$ will be as well. If both are at the Untainted level, then $i'(\texttt{\$b+\$c})$ will end up Untainted.

Integrity violations cannot occur by reading alone, due to the design of the Biba policy. If an object at a higher level reads an object at a lower level, then both objects merely end up at the lower level. A read from a lower level of a higher level causes the lower level to stay at the lower level.

Writes, on the other hand, can cause integrity violations when a write from a lower level to a higher level is attempted. This case will coincide with a taint check failure.

Look to Table 2 again, but this time check Example 2. In this case the programmer is attempting to call a function to create a new directory with the name stored in the variable `$s`. This corresponds to a write from the subject `$s` to the object `mkdir()`.

Recall that `mkdir()` would be classified at the Untainted level because it has the ability to modify the system external to the running program.

Rule 2 of the Biba Model states: $s \in S$ can write to $o \in O$ if and only if $i(o) \leq i(s)$ [5].

If `$s` contained tainted data, an integrity

Example	Expression	Action	Subject	Object	Notes
1	`$o`	*read*	`$o`	`$o`	Basic Read
2	`mkdir( $s )`	*write*	`$s`	`mkdir()`	Function call
3	`$s = $o`	*assign*	`$s`	`$o`	Basic assignment
4	`$b + $c`	*Two* reads*, two* writes *shown in the the following rows:*			
4a	`$b`	*read*	`$b + $c`	`$b`	First variable in the expression
4b	`$c`	*read*	`$b + $c`	`$c`	Second variable in the expression
4c	`$b +`	*write*	`$b`	`+`	First argument to `+`
4d	`+ $c`	*write*	`$c`	`+`	Second argument to `+`

Table 2: Reading and writing under the Biba Integrity Model as applied to taint mode.

violation would occur, corresponding to a taint check failure. This is because the $i(\texttt{\$s}) < i(\texttt{mkdir()})$, and thus `$s` cannot write to `mkdir()`.

Lastly, there is a set of variables that are always at the Untainted security level, and these are the regular expression capture variables `$1`, `$2`, `$3`, and so on. These variables are set by the Perl interpreter automatically after a capture is performed in a regular expression.

By using the *assign* primitive described above and the capture variables, programmers may raise the integrity level of variables as described in Sections 3.5 and 5.2. This is similar to de-classifying documents in the Bell-LaPadula model, though the levels change in the reverse direction.

6 Further Work

6.1 Taint Library

As covered in section 4.2.1, checks done to untaint variables are only as good as the programmer writing those checks. For this reason it would be useful if there were a library of known good taint checks to help increase the security of the general populations programs.

When one was writing a program, they could look up untainting regular expressions from this library and be sure that their implementation was secure. Though therein lies the classical problem: what is secure? There are so many possible uses of perl that might require untainting, it is infeasible to think that a library of these would be possible to build.

However there is a large body of code in perl's CPAN [1], and fledgling programmers have more than enough code to look over

and learn how to do secure programming correctly. With the caveat that they may have to modify it to fit their particular definition of secure, it could be very useful for many programmers.

6.2 Breaking out of Scripting Languages

The only languages that I have come across with taint checks or taint-like features are scripting languages. It would be interesting to look into if this would be possible — or even applicable — for languages such as Java to implement, that compile to bytecode and are run through an interpreter.

As well, implementing a compiler extension for certain languages to track the taintedness of variables could provide some useful information as well, in terms of runtime efficiency and implementation difficulty.

Hacking the compilers and inserting code might not be necessary, however, if tools such as AspectJ were used. A basic taint-checking infrastructure could be built to be used with Java applications, with no changes necessary to the underlying compilers.

Whatever the tool used, the major obstacle to overcome to implement this in non-interpreted languages would be propagating taintedness. In interpreted languages, you have access to every expression as it occurs, so you can just track it as the program executes. See section 3.2 for more on how it is done in perl.

In compiled languages, you don't have the luxury of inspecting expressions at runtime. One way around this would be to try writing a compiler that added an extra line of code near each assignment statement to update a global struct or object that tracks the taintedness for every variable in the program.

This sounds like a very memory and cpu intensive method, and probably wouldn't be useful for anything more than basic quality assurance and testing.

7 Conclusion

No programmer writes (useful) programs free of bugs, and when these bugs can be exploited to break the security of the program, unsatisfactory consequences can occur. Perl's taint mode is a big step in the right direction to help programmers ensure that they write programs free of security related bugs, but it is definitely not a panacea.

Taint mode only checks for certain types of bugs, those that may be triggered by malicious user input. There is still the possibility that the programmer may create temporary files unsafely, save data to the wrong places, or just not write the program according to the specifications. All of these problems can occur whether running under taint mode or not.

Taint mode just gives a little helping hand to help the weary programmer not make as many fatal mistakes. In that sense, this security feature is a resounding success.

References

[1] Cpan - comprehensive perl archive network.

[2] Perl core documentation - perl56delta.

[3] Perl core documentation - perllocale.

[4] Perl core documentation - perlsec.

[5] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley Professional, 2003.

[6] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[7] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. pages 201–220.

[8] Dave Thomas and Andy Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[9] Larry Wall and Mike Loukides. *Programming Perl*. O'Reilly & Associates, Inc., 2000.