Need data from the web - you want Perl

1. The matelook starting point code contains the following Perl:

```
print "<!-- ".join(",", map({"$_='".param($_)."'"} param()))."-->\n";
```

What does it do?

Why is this useful?

Indicate a bug and why it creates a security hole.

Rewrite the code, fixing the bug.

It outputs an HTML comment containing the value of all parameters.
This is useful for debugging because these parameters are usually the key input to your CGI script.

It doesn't check if the value of a parameter contains HTML code.

Displaying HTML code from a malicious user is an easily exploited security hole (http://en.wikipedia.org/wiki/Cross-site_scripting).

This code is simpler and avoids some possible problems:

```
print "<!-- ";
foreach $param (param()) {
    my $value = param($param);
    $value =~ s/\</&lt;/g;
    print "$param='$value' "
}
print "-->\n";
```

2. The validation algorithm used for credit cards is an old one devised by Hans-Peter Luhn (http://en.wikipedia.org/wiki/Hans_Peter_Luhn) (1896-1964), a mathematician at IBM.
The Luhn formula (https://en.wikipedia.org/wiki/Luhn_algorithm) sums all the digits, with weights of 1 for odd positions and 2 for even positions (so every even digit is doubled before adding it to the sum). Odd and even is worked out by counting from the right, as usual. However, if any doubled digit exceeds 9, the two digits of the result are added together, creating a "reduced sum". This reduced sum is added to the total (not the double digit number). For example, a digit 7 in an even position is doubled to make 14, so its contribution to the total is 1+4=5. The credit card number is valid if the total is a multiple of 10.

Here is a (fictitious) VISA card number, entered the way web forms should allow if their developers weren't so lazy: **4564-7953-6021-9047**. Remembering that odd and even positions are counted from the right,

```
Code:    4 5 6 4 7 9 5 3 6 0 2 1 9 0 4 7
Weights: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Sum:     8 5 3 4 5 9 1 3 3 0 4 1 9 0 8 7
(reduced):   *   *   *   *       *
```

and the sum is 70, and 70 mod 10 = 0 as required.

Here is a Python program which takes credit-card numbers as arguments and indicates whether they are not valid or not.

```
#!/usr/bin/python
# written by andrewt@cse.unsw.edu.au as a COMP2041 programming example
# validate a credit card number by calculating its
# checksum using Luhn's formula (https://en.wikipedia.org/wiki/Luhn_algorithm)

import re, sys

def luhn_checksum(number):
    checksum = 0
    digits = reversed(number)
    for (index, digit) in enumerate(digits):
        multiplier = 1 + index % 2
        d = int(digit) * multiplier
        if d > 9:
            d -= 9
        checksum += d
    return checksum

def validate(credit_card):
    number = re.sub(r'\D', '', credit_card)
    if len(number) != 16:
        return credit_card + " is invalid - does not contain exactly 16 digits"
    elif luhn_checksum(number) % 10 == 0:
        return credit_card + " is valid"
    else:
        return credit_card + " is invalid"

if __name__ == "__main__":
    for arg in sys.argv[1:]:
        print(validate(arg))
```

And here is it in action:

```
$ creditcard.py 2389423423423467 9182387723427777 9182380923427773 4564456445644564
2389423423423467 is valid
9182387723427777 is invalid
9182380923427773 is invalid
4564456445644564 is valid
```

1. Discuss how the Luhn Formula is calculated.
2. Why is useful that credit card numbers satisfy this forumla.
3. Discuss how the Python works including why the functions *reversed*, *enumerate* and *int* are used. Also discuss the use of `sys.argv` and `__name__`
4. Discuss how the features of this program can be translated to Perl.

```
#!/usr/bin/perl -w
# written by andrewt@cse.unsw.edu.au as a COMP2041 programming example
# validate a credit card number by calculating its
# checksum using Luhn's formula (https://en.wikipedia.org/wiki/Luhn_algorithm)

sub luhn_checksum {
    my ($number) = @_;
    my $checksum = 0;
    my @digits = reverse(split //, $number);
    foreach $index (0..$#digits) {
        my $digit = $digits[$index];
        my $multiplier = 1 + $index % 2;
        my $check_digit = int($digit) * $multiplier;
        if ($check_digit > 9) {
            $check_digit -= 9;
        }
        $checksum += $check_digit;
    }
    return $checksum;
}

sub validate {
    my ($credit_card) = @_;
    my $number = $credit_card;
    $number =~ s/\D//g;
    if (length $number != 16) {
        return  "invalid - does not contain exactly 16 digits";
    } elsif (luhn_checksum($number) % 10 == 0) {
        return "valid";
    } else {
        return "invalid";
    }
}

foreach $credit_card (@ARGV) {
    print "$credit_card is ", validate($credit_card), "\n";
}
```

3. The following CGI script sends a message to the script author, whose login name is `you`. The message subject and the message body are available via the script parameters `MailSubject` and `MailBody` provided by filling in a form.
Consider for a moment that you are a hacker (yes, I know it's difficult but just pretend :-). What are the potential security problems with such a script that you could exploit? Can you think of different ways they could be exploited (be creative, hackers certainly are)? How can they be overcome?

```
#!/usr/bin/perl
use CGI ':all';
print header, start_html;
$subject = param('MailSubject');
$message = param('MailBody');
if (!open(MAIL, "|mail -s \"$subject\" you")) {
    print h1("Sorry, can't send mail.");
} else {
    print MAIL $message;
    close(MAIL);
}
print end_html;
```

Some of the potential security problems that I know about (there may be others):

- People could use backquotes in the `$subject` variable. Since the `open` is done by passing the `mail` command line to a shell, the backquotes can be used to execute any arbitrary command *before* attempting to run `mail`.
  Example: `$subject == "`/bin/rm -fr .`"` executes the command

  ```
  mail -s "`/bin/rm -fr *`" you
  ```

  which removes a whole bunch of files, and uses the output of the `rm` command (which ought to be the empty string) as the subject of the mail.

  Note: using single quotes in the mail command would protect against back ticks, but not the next exploit.

- People could supply a `$subject` that terminated the subject command line argument, sent the mail to some other address, and then appended some arbitrary Unix commands after a semicolon. Once again, they can execute an arbitrary command as you.
  Example: `$subject == "done\" evil@naughty.com; rm -fr *; echo \"haha tricked"` executes the command

  ```
  mail -s "done" evil@naughty.com; rm -fr *; echo "haha tricked" you
  ```

  which sends mail to `evil@naughty.com` and then removes a whole bunch of files.

- Some versions of `mail` allow you to put a line like `.!`*`command`* to execute commands while you're typing in a message (this is typically called a "shell escape"). People could use the `mail` command's shell escape mechanisms to embed arbitrary commands in the body of the message.
  Example: `$message == "Hello\n.!rm -fr *\nHope you liked the surprise!"` sends the message "Hello\nHope you liked the surprise!", but as a side-effect also executes the `rm` command to remove a whole bunch of files.

To avoid such problems, you would need to make both the `$subject` and `$message` variables "safe" before using them. This could be achieved simply by removing any problematic characters (like backquotes, tildes, exclamation marks, dots). You should also run the script using Perl's `-T` flag (check for "tainted" code), which causes Perl to not even execute your script if you haven't taken enough precautions to avoid (many) potential problems.