Containers Are a Pile of Lies! Part One
Drupal for Symfony Developers
Love/Hate—The Dysfunctional Relationship We Have With Tools

ALSO INSIDE

Artisanal:
Full-Text Searching with Scout

The Dev Lead Trenches:
Coming Aboard!

Community Corner:
The Journey to Becoming a Speaker

Security Corner:
Application-level Data Security

Education Station:
Shifting and Masking with a Side of Crypto

finally():
Blue Collar Coders
We’re hiring PHP developers

15 years of experience with PHP Application Hosting

SUPPORT FOR *php7* SINCE DAY ONE

Contact [careers@nexcess.net](mailto:careers@nexcess.net) for more information.
PHP[TEK] 2018
The Premier PHP Conference
13th Annual Edition
MAY 31ST – JUNE 1ST
Downtown Atlanta

Full schedule announced!
Talks from experts at MailChimp, Salesforce, Etsy, Google, Oracle, Pantheon and many more!

tek.phparch.com
Features

3 Containers Are a Pile of Lies! Part One
Larry Garfield

10 Drupal for Symfony Developers
Antonio Perić-Mažar

22 Love/Hate—The Dysfunctional Relationship We Have With Tools
Shahina Patel

Columns

2 Know Your Tools

26 Artisanal: Full-Text Searching with Scout
Joe Ferguson

29 Security Corner: Application-level Data Security
Eric Mann

34 The Dev Lead Trenches: Coming Aboard!
Chris Tankersley

38 Community Corner: The Journey to Becoming a Speaker
James Titcomb

40 January News

41 Education Station: Shifting and Masking with a Side of Crypto
Edward Barnard

48 finally{}: Blue Collar Coders
Eli White

Editor-in-Chief: Oscar Merida
Editor: Kara Ferguson

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Brzez, Oscar Merida, Sandy Smith

php[architect] is published twelve times a year by musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Although all possible care has been placed in assuring the accuracy of the contents of this magazine, including all associated source code, listings and figures, the publisher assumes no responsibilities with regards of use of the information contained herein or in all associated material.

php[architect], php[a], the php[architect] logo, musketeers.me, LLC and the musketeers.me, LLC logo are trademarks of musketeers.me, LLC.

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com
Print ISSN 1709-7169
Digital ISSN 2375-3544
Copyright © 2018—musketeers.me, LLC
All Rights Reserved
Shifting and Masking with a Side of Crypto

Edward Barnard

The basics can be tricky. This month we take a careful walk-through of a few lines of cryptographic code in PHP. This leads us through the difference between ones’ complement and two’s complement representation. We achieve weirdness by combining logical AND with integer addition.

Thirty-five years ago Ed Post of Tektronix¹ expressed the insecurities we all felt as computer programmers. The previous five years had produced the Star Wars trilogy, but more importantly, those five years produced personal computers.

In 1978, just two years after it was founded, Apple won a contract with the Minnesota Education Computing Consortium to supply 500 computers for schools in the state. MECC had developed a sizable catalog of educational software (including the iconic Oregon Trail) which it made freely available to Minnesota schools. Soon the MECC floppy disks and Apple II’s became popular elsewhere across the country. As Steve Jobs said in a 1995 oral history interview with The Computerworld Smithsonian Awards Program:

“One of the things that built Apple II’s was schools buying Apple II’s.”

How Steve Jobs Brought the Apple II to the Classroom²

How is this a problem for professional computer programmers? Looking back from a generation later, the reason is not obvious. Computer programmers were highly paid with excellent job security. The only computers available were the million-dollar mainframes, although departmental minicomputers were becoming available for mere tens of thousands of dollars. A computer programmer was very much like a rocket scientist since one requires a rocket to be a rocket scientist, and one requires a mainframe to be a computer programmer.

A high school graduate (or dropout) could, for example, become an excellent auto mechanic, but not a rocket scientist—or computer programmer. Steve Jobs and Apple changed it all.

Real Programmers

Last month’s Education Station asked “What is a Real Programmer?” without directly answering the question. Ed Post’s parody essay Real Programmers Don’t Use PASCAL³ tells us the answer regarding those job insecurities of a generation ago:

But, as usual, times change… anyone can buy and even understand their very own Personal Computer. The Real Programmer is in danger of becoming extinct, being replaced by high-school students with TRASH-80s!

How is this a problem for professional computer programmers? Looking back from a generation later, the reason is not obvious. Computer programmers were highly paid with excellent job security. The only computers available were the million-dollar mainframes, although departmental minicomputers

Programmers on their staff with 12 year old Pac-Man players (at a considerable salary savings). [Emphasis mine.]

This famous bit of hacker folklore harkens back to the days of glory, or so our elders would have us believe. We’ve all heard the stories of when our elders had to walk ten miles to school every day, barefoot in the snow, uphill both ways. The fears for our profession were real. But how is this going to help us today?

Keypunching (see Figure 1) has gone the way of the electric typewriter. How can it possibly be relevant? It isn’t. But there is one related skill which we’re going to see today: desk checking. We’ll be using desk checking to walk through our cryptography code in PHP.

Figure 2 is a sample program from the original Fortran Programmer’s Reference Manual⁴ published in 1956. It is set up to be keypunched onto a card deck. Each line will become one 80-column punched card.

Bernard Smith⁵ explains this problem of the late 1970s:

Input to the mainframe was made using a tray of punch cards… and put my tray through a small window… I did not know when my particular program would be run. I remember working on the basis of 4 hours, and hoping I could get

1 Ed Post of Tektronix: http://phpa.me/real-programmers-pascal
2 How Steve Jobs Brought the Apple II to the Classroom: http://phpa.me/hackedu-steve-jobs
3 Real Programmers Don’t Use PASCAL: http://phpa.me/ryerson-real-programmers
5 Bernard Smith: http://phpa.me/bsmith-my-computers
To put this in perspective, imagine developing a complex web page. It takes four hours to load (on a good day), so you get to see it once by mid-morning, once by mid-afternoon, and you might sneak one more peek overnight. After loading your web page, you’ll want to edit your code. Your code, of course, exists as a tray of punch cards. So, to modify your code, you need access to the keypunch in some other building. Figure 3 shows a typical keypunch set aside for programmers’ use. The sign on the wall reads, “Express Keypunch: 3 minutes or 6 cards (or until asked to leave!)”

Picture a typical program of, say, one thousand lines of code, handwritten on 40 pages of coding forms. You would turn it into the production keypunch queue. Take another look at Figure 1, which shows production keypunching at Texas A&M University. Your program is typed twice. The cards are punched on the machine to the right (the “card punch”), and then they are typed again at the machine on the left (the “card verifier”). The program will be precisely as you hand wrote it, typos and all.

You can see why desk checking was one of the skills taught in computer engineering classes in college. Computer time was an extremely scarce resource. (Imagine your professional consternation at kids having an Apple II available in high school when you were lucky to run your program ten or twelve times in an entire week!)

Desk checking begins as simple proofreading. Specifically, reading back through the stack of handwritten coding forms, carefully looking for any errors. Picture a keyboard with no backspace or delete key, ever.

But wait! There’s more! Remember the inverted time factor. When you only get access to the computer once or twice a day, you’ll spend the rest of the day preparing to make the most of this access. Unlike today, you are not the critical resource. The mainframe computer is.

A real desk check is a structured walkthrough on paper. We take a sample set of data and run through the entire program, acting as the computer, using pencil and paper (and possibly slide rule). This is the skill which remains valuable and which we’ll be learning today. This is a skill practiced by every Real Programmer before he or she walked to the computer center (ten minutes, uphill both ways).

Logical and Bitwise Operators

Before we proceeded, for reference the table below summarizes PHP’s logical and bitwise operators are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a &amp; $b</td>
<td>Bitwise And</td>
<td>Bits set in two variables</td>
</tr>
<tr>
<td>$a</td>
<td>$b</td>
<td>Bitwise Or</td>
</tr>
<tr>
<td>$a ^ $b</td>
<td>Bitwise Exclusive Or</td>
<td>Bits set in one but not both variables</td>
</tr>
<tr>
<td>~$a</td>
<td>Bitwise Not</td>
<td>Reverses the bit settings</td>
</tr>
<tr>
<td>$a &lt;&lt; $b</td>
<td>Shift Left</td>
<td>Shifts the bits in $a by $b steps to the left</td>
</tr>
<tr>
<td>$a &gt;&gt; $b</td>
<td>Shift Right</td>
<td>Shifts the bits in $a by $b steps to the right</td>
</tr>
<tr>
<td>$a &amp; $b</td>
<td>Logical And</td>
<td>True if both are true</td>
</tr>
<tr>
<td>$a</td>
<td>$b</td>
<td>Logical Or</td>
</tr>
<tr>
<td>$a xor $b</td>
<td>Logical Exclusive Or</td>
<td>True if only one is true</td>
</tr>
<tr>
<td>!$a</td>
<td>Logical Not</td>
<td>True is $a is not true, false if $a is true</td>
</tr>
</tbody>
</table>

Note there are also logical and and or operators which have a lower precedence than && and ||.

Real-World PHP Cryptography

Scott Arciszewski pointed me to a useful example of shifting and masking in PHP. But, why do we care? Does shifting and masking help us build web pages? To investigate the question of “do we care,” I attended the Zend PHP Certification

---

1 CC BY 2.0: [http://creativecommons.org/licenses/by/2.0](http://creativecommons.org/licenses/by/2.0)

6 bitwise: [http://php.net/language.operators.bitwise](http://php.net/language.operators.bitwise)

7 Scott Arciszewski: [https://twitter.com/CiPHPerCoder](https://twitter.com/CiPHPerCoder)
Shifting and Masking with a Side of Crypto

Listing 1

```php
1. <?php
2. declare(strict_types=1);
3. namespace ParagonIE\ConstantTime;
4. /**
5. * Copyright 2014 Steve "Sc00bz" Thomas (steve at ...)
7. * Permission is hereby granted...
8. */
9. class Hex {
10.    public static function encode(string $bin_string): string {
11.        $hex = ''; // initialize output string
12.        $len = Binary::safeStrlen($bin_string); // get length of string
13.        for ($i = 0; $i < $len; ++$i) {
14.            $chunk = Binary::safeSubstr($bin_string, $i, 2);
15.            $b = $chunk[1] >> 4;
16.            $c = $chunk[1] & 0xf;
17.            (87 + $b + ((($b - 10) >> 8) & ~38)),
18.        } // end for loop
19.        return $hex;
20.    }
21.}
```

This "divide and conquer" analysis is an important desk-checking skill. We can focus in on each of those sections, in order, without being distracted by what's happening in the other lines of code.

In the real world, of course, we have code editors and step debuggers to do this sort of thing for us. There are times, though, when it takes a human to grok the situation. Such a human is showing the casual skills of a "Real Programmer."

Definition15: The computer folklore term Real Programmer has come to describe the archetypical "hardcore" programmer who eschews the modern languages and tools of the day in favor of more direct and efficient solutions—closer to the hardware.

Lines 18-19

Lines 18-19 unpack our ASCII “Q” into $chunk. What actually happens? Let’s check the online documentation for unpack16. Format C says the input is an unsigned char (8-bit character). This sets $chunk to 0x51, which is ASCII “Q.”

We can guess Binary::safeSubstr() extracts our character in a binary-safe fashion. A check of the library code17 shows

---

8 Zend PHP Certification Boot Camp: https://join.in/talk/9e2dd
9 Christian Wenz: https://twitter.com/chwenz
10 the list of exam topics: http://phpa.me/zend-php-cert
11 Constant-Time Character Encoding: http://phpa.me/pargonie-const-time-encode
12 Anthony Ferrara: https://twitter.com/ircmaxell
13 It’s All About Time: http://phpa.me/ircmaxell-about-time
14 ASCII table: http://www.asciitable.com/
15 Definition: http://phpa.me/real-programmers-pascal
17 library code: http://phpa.me/github-Binary-php

---
it's a wrapper for PHP's `substr()` or `mb_substr()`.

I don't know why we use two characters as input, rather than a single character. I can speculate it has to do with allowing for different endianness\(^{18}\), but we'd have to ask the author.

**Line 20**

Line 20 $c = \text{chunk}[1] \& 0xf$; is pure PHP, but what does it mean? Why use $\text{chunk}[1]$ rather than $\text{chunk}[0]$? The answer is buried in the notes\(^{19}\) for PHP `unpack()`:

---

**Caution** If you do not name an element, numeric indices starting from 1 are used. Be aware that if you have more than one unnamed element, some data is overwritten because...

---

Well, then. I don't write `unpack()` code very often, and likely you don't either. I didn't know one can use `unpack()` in such a way it overwrites its own output. (Awesome (sarcasm).) When we have something which looks off, in this case, an array starting with 1 rather than 0, it pays to take a close look at the documentation (if any).

The & in line 20 says we are doing a bitwise AND. Last month's *Education Station* presented the truth table for AND:

<table>
<thead>
<tr>
<th>Left operand</th>
<th>Right operand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Our ASCII table told us "Q" is 0x51, or 0101 0001 in binary. The same table would tell us 0xf is 1111 in binary. Or, expanding to 8 bits, 0xf is 0000 1111 in binary. Let's do the calculation.

\[ 0101 \ 0001 \ \text{Schunk}[1] \implies 0x51 \implies "Q" \]
\[ 0000 \ 1111 \ 0xf \]
\[ \text{ } \]
\[ 0000 \ 001 \ \text{Schunk}[1] \& 0xf; \]

---

**Line 21**

Line 21 $b = \text{chunk}[1] >> 4$; tells us to perform an arithmetic shift right of four bits. The PHP bitwise operators\(^{20}\) documentation tells us "Right shifts have copies of the sign bit shifted in on the left, meaning the sign of an operand is preserved."

\[ 0101 \ 0001 \ \text{Schunk}[1] \implies 0x51 \]
\[ >> 4 \text{ Shift 4 bits to the right} \]
\[ 0000 \ 0101 \ 0x05 \implies 5 \text{ decimal} \]

Finally! We have our starting point from lines 18-21:

\[ \text{Schunk} = 0x51; \text{ // ASCII 'Q'} \]
\[ \text{$c = 0x1; \text{ // Right-most (lower) hex digit} \text{}} \]
\[ \text{$b = 0x5; \text{ // Upper hex digit} \text{}} \]

---

\(^{18}\) endianness: https://en.wikipedia.org/wiki/Endianness

\(^{19}\) notes: http://phpa.me/php-unpack-notes

\(^{20}\) bitwise operators: http://php.net/language.operators.bitwise

---

**Ones' Complement**

Before we tackle the unmitigated trickiness of our constant-time encoding algorithm (lines 24-25), we need to review ones' complement and two's complement arithmetic. Complements are both simple and weird, which by definition makes this fun!

In PHP, 0 produces an integer which, in binary, is all ones. 0, in binary, is all zeroes (as you would expect), and PHP's bitwise NOT operator ~ says to flip all the bits, meaning, change every 0 bit to a 1 and change every 1 bit to a 0. How many bits is “all the bits”? It depends on the word size of your computer. On a 32-bit machine you'd have 32 bits, and on a 64-bit machine, your result (all zeros becoming all ones) will be a 64-bit integer.

For computing, a word is a fixed-size unit of data, or set of bits, worked on by the processor. The word size affects things like how many characters are supported, the largest numbers which can be represented, and how memory is addressed.

This is where the weirdness comes in. The bitwise operators &, |, ^, ~, <<, and >> operate on PHP integers. They operate on the underlying value—the binary representation—contained in the integer. Wikipedia’s Ones’ complement\(^{21}\) article includes a table to help explain the distinction. See the tables below.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Logical Value</th>
<th>Arithmetic Value</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8-bit)</td>
<td>(unsigned)</td>
<td>(ones’ complement)</td>
<td></td>
</tr>
<tr>
<td>0111 1111</td>
<td>127</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>0111 1110</td>
<td>126</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0000 0010</td>
<td>2</td>
<td>2</td>
<td>&lt;= see below</td>
</tr>
<tr>
<td>0000 0001</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td>0</td>
<td>0</td>
<td>&lt;= &quot;plus zero&quot;</td>
</tr>
<tr>
<td>1111 1111</td>
<td>255</td>
<td>-0</td>
<td>&lt;= &quot;minus zero&quot;</td>
</tr>
<tr>
<td>1111 1110</td>
<td>254</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>1111 1101</td>
<td>253</td>
<td>-2</td>
<td>&lt;= see below</td>
</tr>
<tr>
<td>1000 0001</td>
<td>129</td>
<td>-126</td>
<td></td>
</tr>
<tr>
<td>1000 0000</td>
<td>128</td>
<td>-127</td>
<td></td>
</tr>
</tbody>
</table>

For example, 2 and -2 show the bits flipped on each other: 0000 0010 2 2 1111 1101 253 -2

The PHP operators are fine. They are not weird. They each do bit manipulations as expected. What's weird is the number representation. In Listing 2, we call the pattern of all zeroes “plus zero”. “Plus zero” sounds a bit redundant, given zero is

\(^{21}\) Ones’ complement: http://phpa.me/wikip-ones-complement
zero, and since the number is all zeroes, we're okay.

What's weird is we call the pattern of all ones "minus zero." Do we have to call it that? Listing 2 shows the integer sequence as we count down: 2, 1, 0, -1, -2. It's often used as a method to represent signed numbers and preferable to one's you think about it, there can only be one integer between 1 and 3. It would be 2. In the same way, there can only be one integer between +1 and -1. This would be zero. We do in fact have two different one's complement representations of zero: the all-zeroes pattern, and the all-ones pattern. Weird.

**Signed Zero**

You're surely wondering whether we have to care the one's complement notation includes something weird called "minus zero." We do; let's take a look.

The first successful supercomputer, the CDC 6600[^22], used one's complement arithmetic. Programmers had to allow for both "plus zero" and "minus zero". It was a great source of coding errors. Modern computers have adopted a better system of integer arithmetic, which means we do **not** have to care. This is a good thing!

However, there is a catch. "The IEEE 754 standard for floating-point arithmetic (presently used by most computers and programming languages supporting floating-point numbers) requires both +0 and -0." Signed zero[^23] Now you're aware "minus zero" does matter when working with floating-point numbers, we won't mention it further.

**Two's Complement**

Two's complement[^24] notation solves the "minus zero" problem:

Compared to other systems for representing signed numbers (e.g., one's complement), two's-complement has the advantage that the fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers (as long as the inputs are represented in the same number of bits, and any overflow beyond those bits is discarded from the result). This property makes the system simpler to implement, especially for higher-precision arithmetic. Unlike one's-complement systems, two's complement has no representation for negative zero, and thus does not suffer from its associated difficulties.

Conveniently, another way of finding the two's complement of a number is to take its one's complement and add one: the sum of a number and its one's complement is all '1' bits, or $2^N - 1$; and by definition, the sum of a number and its two's complement is $2^N$.

In short, PHP uses one's complement for **logical** operations (the bitwise operators), but uses two's complement for **arithmetic** operations such as adding two integers. When we combine these two modes in the same expression, analysis is going to be tricky. Which is precisely our situation with lines 24 and 25.

Let's see how this works. Using Listing 2, let's convert 2 to -2:

```
0000 0010 Arithmetic 2
1111 1110 One's complement
1111 1100 Add 1 to form two's complement
```

I adapted Wikipedia's Two's complement[^25] table to show how this works.

<table>
<thead>
<tr>
<th>Original</th>
<th>Binary</th>
<th>Negate</th>
<th>Two's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000</td>
<td>0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001</td>
<td>-1</td>
<td>1111 1111</td>
</tr>
<tr>
<td>2</td>
<td>0000 0010</td>
<td>-2</td>
<td>1111 1110</td>
</tr>
<tr>
<td>126</td>
<td>0111 1110</td>
<td>126</td>
<td>1000 0010</td>
</tr>
<tr>
<td>127</td>
<td>0111 1111</td>
<td>127</td>
<td>1000 0001</td>
</tr>
<tr>
<td>-128</td>
<td>1000 0000</td>
<td>-128</td>
<td>1000 0000</td>
</tr>
<tr>
<td>-127</td>
<td>1000 0001</td>
<td>127</td>
<td>0111 1111</td>
</tr>
<tr>
<td>-126</td>
<td>1000 0010</td>
<td>126</td>
<td>0111 1110</td>
</tr>
<tr>
<td>-2</td>
<td>1111 1110</td>
<td>2</td>
<td>0000 0010</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111</td>
<td>1</td>
<td>0000 0001</td>
</tr>
</tbody>
</table>

When we're doing both arithmetic and logical operations on the same integers:

- Arithmetic operations use two's complement arithmetic
- Logical operations use one's complement bit manipulation

Let's do some code.

**Line 24**

Let’s calculate \((87 + \$b + (((\$b - 10) >> 8) & ~38))\) by starting at the innermost expression and working our way out. \$b is 5, which is 0b101, or 0000 0101 in binary. Therefore \(\$b - 10\) is 5 - 10, which is -5. We just learned -5, in PHP, is in two's complement representation. Which means we take 5, in binary (0000 0101), flip the bits, and add 1. However, we need to extend our calculation to 16 bits. You'll see why at the next step:

```
0000 0000 0000 0101 5
1111 1111 1111 1010 -5 (flip the bits)
+1 Add one
```

Our next instruction (((\$b - 10) >> 8)) is to shift the result 8 bits to the right. Remember right shifts propagate the sign

[^23]: Signed zero: [http://phpa.me/wikip-signed-zero](http://phpa.me/wikip-signed-zero)
[^24]: Two's complement: [http://phpa.me/wikip-twos-complement](http://phpa.me/wikip-twos-complement)
[^25]: Two's complement: [http://phpa.me/wikip-twos-complement](http://phpa.me/wikip-twos-complement)
bit. We're doing the calculation as a 16-bit integer, but the result would be the same for a 32-bit or 64-bit word size. We therefore lose the bottom, or rightmost, 8 bits (1111 1011) and replace them with the next 8 bits from the left (1111 1111) or as 16 bits (1111 1111 1111 1111).

Next comes ~38. Form the ones' complement of decimal 38, which is hexadecimal 0x26:

0000 0000 0100 0110 0x26 === 38 decimal
1111 1111 1011 1001 One's complement

Our next calculation is (((b - 10) >> 8) & ~38). We've done the calculations, so we know this reduces to a bitwise AND:

1111 1111 1111 1001 Prior result
0000 0000 0010 0110 Flip the bits +1 Add 1
-------------------
0000 0000 0010 0111 0x27 === 39 decimal

Given the two's complement of (1111 1111 1101 1001) is 39, this tells us (1111 1111 1101 1001) is decimal -39. Our calculation is now (92 + -39) which equals 53.

Let's quickly summarize the steps:
1. b - 10: 5 - 10 === -5
2. -5 shifted right 8 bits === -1
3. -38 === -39
4. -1 (which is all ones) AND anything is anything. -1 & -39 === -39
5. 87 + 5 + -39 === 53

Remember we are encoding ASCII "Q" as a string of hexadecimal digits. "Q" is hexadecimal 51. By a remarkable and happy coincidence, chr(53) === 'S'.

Fast Forward
We noted lines 24 and 25 are identical except line 24 does the calculation for $b and line 25 does the calculation with $c. Line 24 began with $b having the value 5 and calculated the result as 53.

When we evaluated lines 20-21 we noted $b is 5 and $c is 1. Given 1 ($c) is four less than 5 ($b), we can infer line 25 will calculate to 49. And, sure enough, ASCII '1' is decimal 49.

Pack It In
Line 22 now reduces to:

\$hex .= pack('CC', 53, 49);

Since we know there will only be one trip through the for loop, and $hex began as an empty string, we know the above line produces our final result. pack('CC', ...) produces the ASCII characters corresponding to decimal 53 and decimal 49. 53 is ASCII '5'; 49 is ASCII '1'. Thus we have:

\$hex .= '51';

Summary
When we need to take a careful look at how a piece of code works, we can draw on the ancient art of desk checking. We can be thankful its day-to-day necessity has gone the way of the keypunch. However, you can use this technique "paper prototype" or "whiteboard" a more complicated algorithm before you and your team dive into your code editors by walking for how it should work and what data (variables) you'll need to track.

We walked through a constant-time-encoding library for PHP cryptography. We saw:

Hex::encode('Q') === '51';
ord( 'Q' )  === 0x51;

We distinguished between the PHP bitwise operators, which work with ones' complement representation, and the PHP arithmetic operators, which work with two's complement representation. We found it's perfectly valid, though weird, to combine the two types of operations in a single PHP expression.

The term “Real Programmer” comes from a tongue-in-cheek parody essay. It's intended to be humorous rather than to be taken seriously. A Real Programmer searches for weirdness in "The Basics."
Borrowed this magazine?

Get php[architect] delivered to your doorstep or digitally every month!

Each issue of php[architect] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, e-commerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions
Starting at $49/Year

http://phpa.me/mag_subscribe