Writeup by Razvi
RazviOverflow Twitter Github

Integer Operations

February the 20th 2020

Last edited on April 8, 2020

Contents

1. Introduction

Disclaimer: *I assume you solved or at least tried to solve the binary before reading this document. Its main purpose is purely educational and it is intended to help you learn and further your knowledge.*

In this first chapter we will see what this binary is all about and introduce the main concepts needed in order to solve it.

1.1 Challenge description

Title: Integer Operations

Description: You are back to school learning about addition, subtraction and equations with integers. You'll have to fully understand how they behave in order to pass the exam.

The name of the binary/challenge as well as its description are a good hint and starting point. *Integer operations* and *how integers behave* are clear references to the well-known vulnerability/software bug ***Integer Overflow***.

1.2 Integer Overflow

There is a lot of info about integer overflows/underflows, why the occur in memory, how to exploit them, how to avoid them, etc... But I consider the following references as the main ones to start understanding it:

1. Basic Integer Overflow by Blexim. Published in Phrack#60 [1].
2. Understanding Integer Overflows in C/C++ by Dietz, Li, Regehr and Adve [2].
3. Integer Overflow overview autogenerated by Science Direct [3].
4. Integer Overflow entry on Wikiedia [4]. Even though Wikipedia has bad reputation, the entry about Integer Overflows is pretty good and very useful to read discover other references and resources (bottom of the page).

Broadly speaking, integer overflows/underflows occur when a given number is stored in a certain data type that it does not fit into. That is, a number bigger than the maximum integer (`int`) value or a number lesser than the minimum integer value pretended to be assigned to a `int` variable. They can also appear when signed data types are treated as unsigned or viceversa.

Integer overflows are a type of numerical software errors/bugs. "*These errors include overflows, underflows, lossy truncations (e.g., a cast of an int to a short in C++ that results in the value being changed), and illegal uses of operations such as shifts (e.g., shifting a value in C by at least as many positions as its bitwidth).*" [2].

Back in 2002, Blexim categorized these numerical errors into: 1) Integer overflows and 2) Signedness bugs [1]. According to him/her, "*an integer overflow is the result of attempting to store a value in a variable which is too small to hold it*". Two types of integer overflows were described:

- Widthness overflows. They occur when a variable of a given size (width) like `int` which is typically 32 bits long, is assigned to a smaller variable like `short`, which is typically 16 bits long. Blexim also talks about *truncation* and *integer promotion/demotion* that are quite important concepts.
- Arithmetic overflows. They occur when the result of an arithmetic operation is bigger than the data type of the variable it will be stored into.

On the other hand, signedness bugs "*occur when an unsigned variable is interpreted as signed, or when a signed variable is interpreted as unsigned. This type of behaviour can happen because internally to the computer, there is no distinction between the way signed and unsigned variables are stored.*"

As it was aforementioned, there is plenty of literature about integer overflows. It is a well known vulnerability/bug and, as such, there are plenty of public recommendations to deal with it. The Common Weakness Enumeration[1] (CWE) defines various weaknesses that involve integer overflows or other types of numerical errors. Table 1.1 shows some of these weaknesses (there are many more in the CWE's INT category[2]). Moreover, the weakness Integer Overflow or Wraparound (CWE-190) is placed 8th in the 2019 CWE Top 25 Most Dangerous Software Errors [5] ranking.

Table 1.1: CWE Weaknesses involving integer and numerical errors. (Source: CWE CATEGORY: INT)

Identifier	**Title**
CWE-190	Integer Overflow or Wraparound
CWE-191	Integer Underflow or Wraparound
CWE-192	Integer Coercion Error
CWE-194	Unexpected Sign Extension
CWE-195	Signed to Unsigned Conversion Error
CWE-197	Numeric Truncation Error
CWE-680	Integer Overflow to Buffer Overflow

[1] `https://cwe.mitre.org/index.html`

[2] `https://cwe.mitre.org/data/definitions/1158.html`

2. Solving the challenge

In order to solve the challenge you will only need:

- Understanding how integer overflows and truncation work in memory (at bit level).
- Understanding the size of primitive data types matter and, even though `int`'s size may vary, on most modern compilers it is 32 bits [6, 7].
- A calculator with bit-view mode, and preferably bit-toggling capabilities, in order to help you visualize what is actually happening. Windows10's default calculator in Programmer mode/view is awesome. As a Linux alternative you can use KCalc[1].

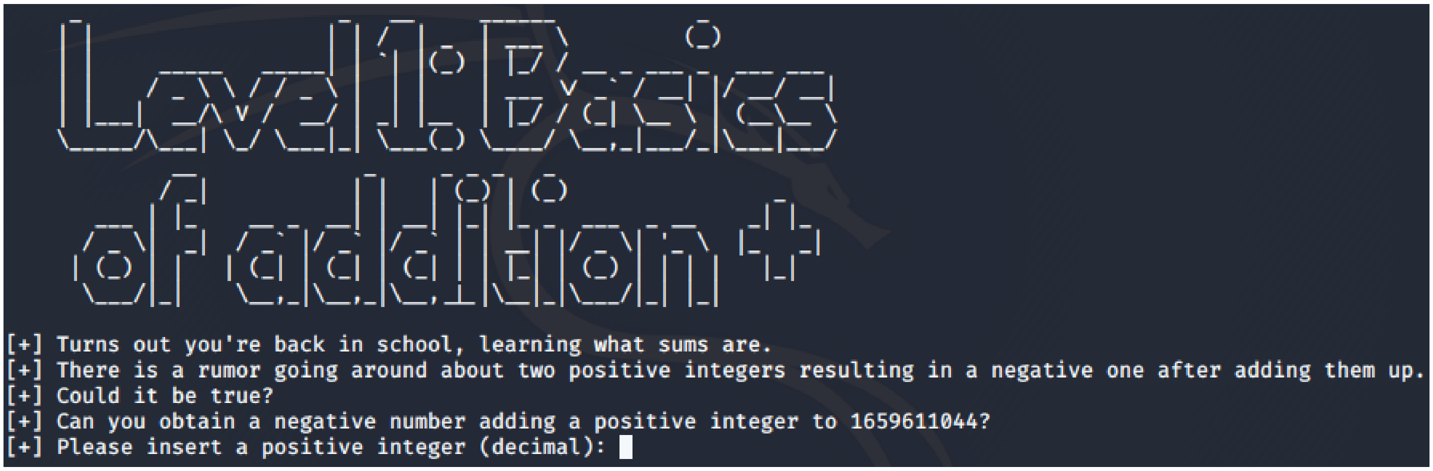
```
By Razvi

[!]
[******] PLEASE KEEP IN MIND:
[******]         If you fail a level you must start all over again.
[******]         Save your correct answers so you can insta write them.
[!]
```

Figure 2.1: Binary's introduction.

[1] https://kde.org/applications/utilities/org.kde.kcalc

2.1 Level 1: Basics of addition



```
[+] Turns out you're back in school, learning what sums are.
[+] There is a rumor going around about two positive integers resulting in a negative one after adding them up.
[+] Could it be true?
[+] Can you obtain a negative number adding a positive integer to 1659611044?
[+] Please insert a positive integer (decimal): 
```

Figure 2.2: Formulation of the first level.

The level's description states that two positive integers could result in a negative one after adding them up. It sounds like a signedness bug resulting from an arithmetic operation, i.e., the addition.

Every time you run the binary it will ask you about a different, random number. To pass the level you can either provide a solution of that particular number or look for a general solution that works with every single time. So far, the only thing we know for sure is that the program will give you a positive integer, in the example shown in Figure 2.2 it is 1659611044, and you must insert another positive one. So far we can assume that internally the program is working with `signed int` data types because otherwise a negative result would be impossible.

As you may already know, when working with `signed int` the most significant bit (the leftmost bit) is the one that indicates the sign. If it is set, the number is negative. Figure 2.3 shows the bit representation of both a positive (a) and a negative (b) number.

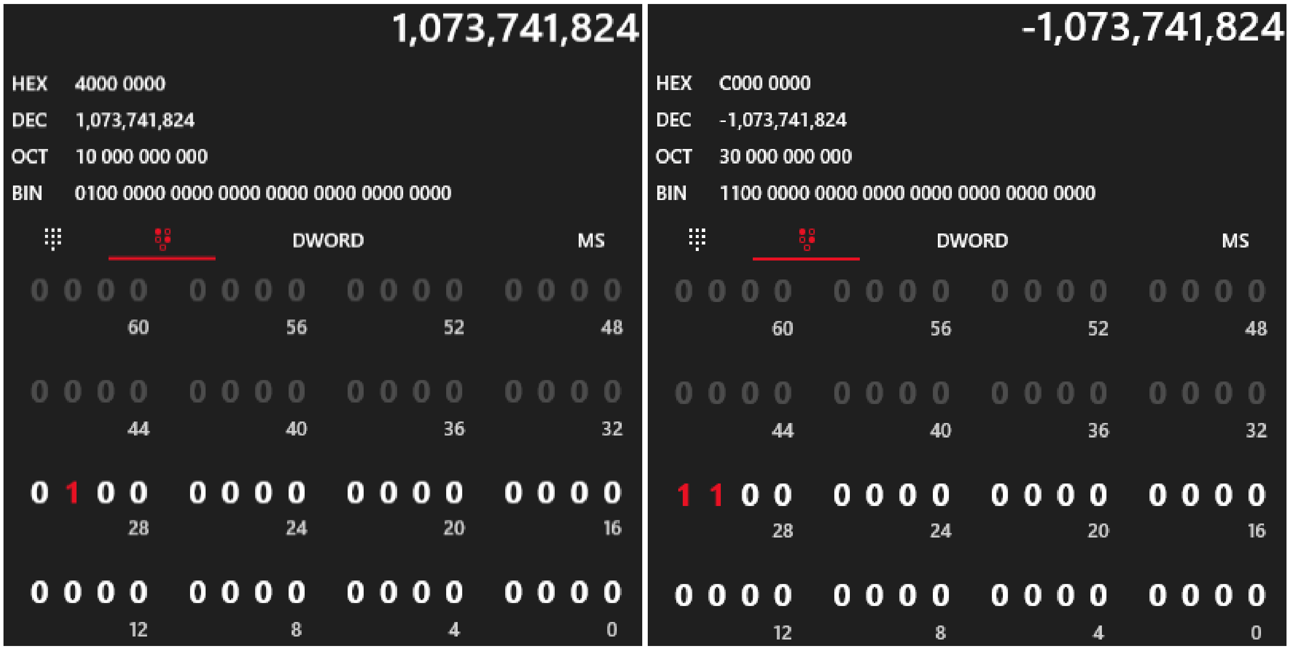
If you want to further read about how the CPU deals with positive and negative numbers you must read about ***Two's complement*** [8, 9].

In order to solve the level you must realize that the most significant bit must be set after the addition. That is, insert a number that added to the one given by the program results in any other integer whose most significant bit is set. Since the program will always generate a positive integer to be added with, there is a general solution for this level. That number is:

Decimal: 2147483647
Hex: 0x7FFFFFFF
Binary: Figure 2.4

It works no matter the positive integer given by the program because of the binary representation of the number 2147483647. It does not matter what other positive integer in the range $[1, 2147483647]$ is added to it, it will always result in a number with the 31st bit set, that is, a negative number. *Please remember an integer `int` is 32 bits long, hence the max value 2147483647.*

You can write 2147483647 as the answer for the first level and it will always work. This number is `INT_MAX` since it is the maximum positive value that can be stored in a `signed int` variable.



1,073,741,824

HEX 4000 0000
DEC 1,073,741,824
OCT 10 000 000 000
BIN 0100 0000 0000 0000 0000 0000 0000 0000

```
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     44        40        36        32
0 1 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     28        24        20        16
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     12         8         4         0
```

-1,073,741,824

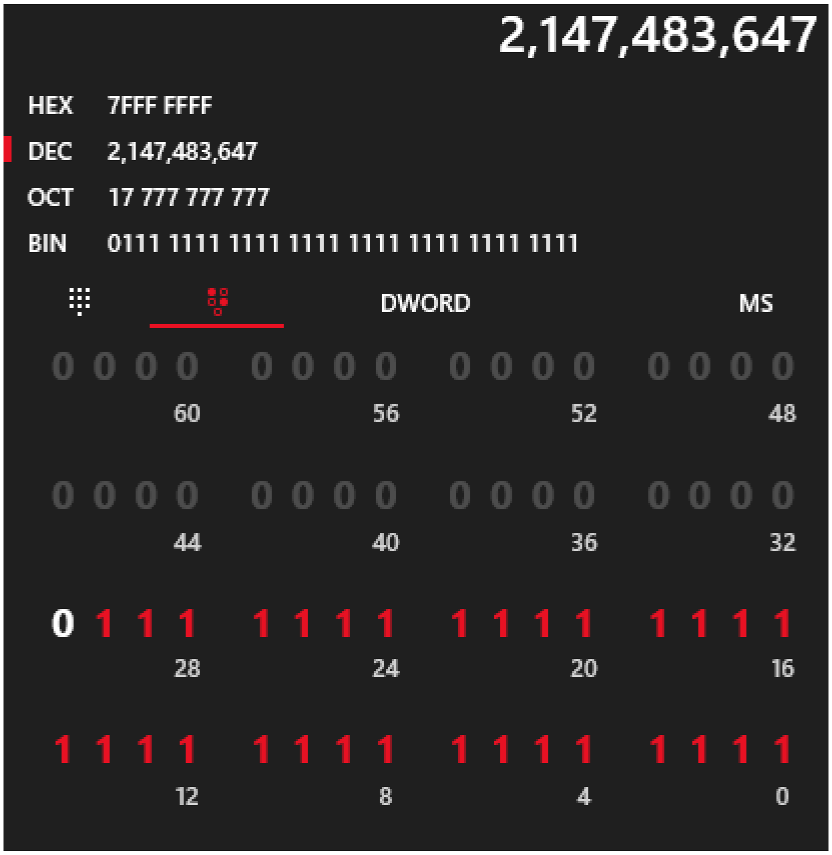
HEX C000 0000
DEC -1,073,741,824
OCT 30 000 000 000
BIN 1100 0000 0000 0000 0000 0000 0000 0000

```
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     44        40        36        32
1 1 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     28        24        20        16
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     12         8         4         0
```

(a) Positive integer

(b) Negative integer

Figure 2.3: 32 bit representation of integer



2,147,483,647

HEX 7FFF FFFF

DEC 2,147,483,647

OCT 17 777 777 777

BIN 0111 1111 1111 1111 1111 1111 1111 1111

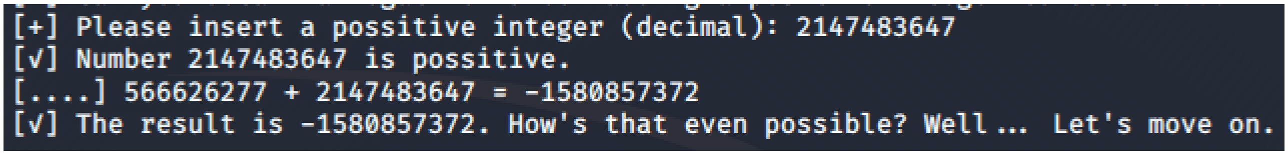
```
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
      60        56        52        48

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
      44        40        36        32

0 1 1 1   1 1 1 1   1 1 1 1   1 1 1 1
      28        24        20        16

1 1 1 1   1 1 1 1   1 1 1 1   1 1 1 1
      12         8         4         0
```

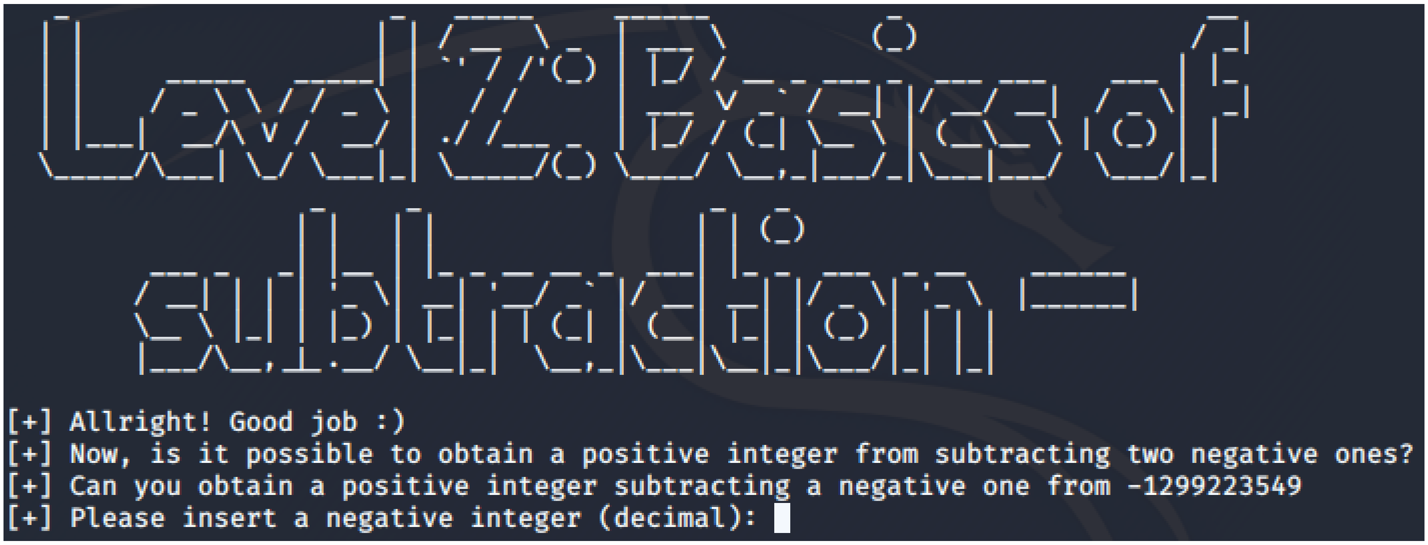
Figure 2.4: General solution to level 1.



```
[+] Please insert a possitive integer (decimal): 2147483647
[√] Number 2147483647 is possitive.
[....] 566626277 + 2147483647 = -1580857372
[√] The result is -1580857372. How's that even possible? Well ... Let's move on.
```

Figure 2.5: Solving the first level.

2.2 Level 2: Basics of subtraction



```
[+] Allright! Good job :)
[+] Now, is it possible to obtain a positive integer from subtracting two negative ones?
[+] Can you obtain a positive integer subtracting a negative one from -1299223549
[+] Please insert a negative integer (decimal):
```

Figure 2.6: Formulation of the second level.

The second level is exactly the opposite to the first one, i.e., getting a positive number from the subtraction of two negative integers. Just like the first level, every time you run the binary the program will give you a different negative integer to subtract from. Just like before, you can solve the second level by either enter a solution for the particular case or finding a general solution that works with every single negative integer.

Once again, it is of vital importance to think about the numbers in their binary form. Keep in mind that, as seen in Figure 2.3b, a negative number has the most important bit set. In the context of the binary, since we are dealing with `int`s, that is the 31st bit set. We know the program will always generate a negative random number. That means the number we must subtract a negative integer from will always have the 31st bit set. What we must find now is an integer that added to the one provided by the program results in number whose 31st bit is not set. Before continuing please notice I said *added* and not subtracted because I find it easier to understand the concepts thinking about sum of bits rather than subtraction. It is equally valid to talk about summing two negative integers since the following are two equivalent arithmetic operations:

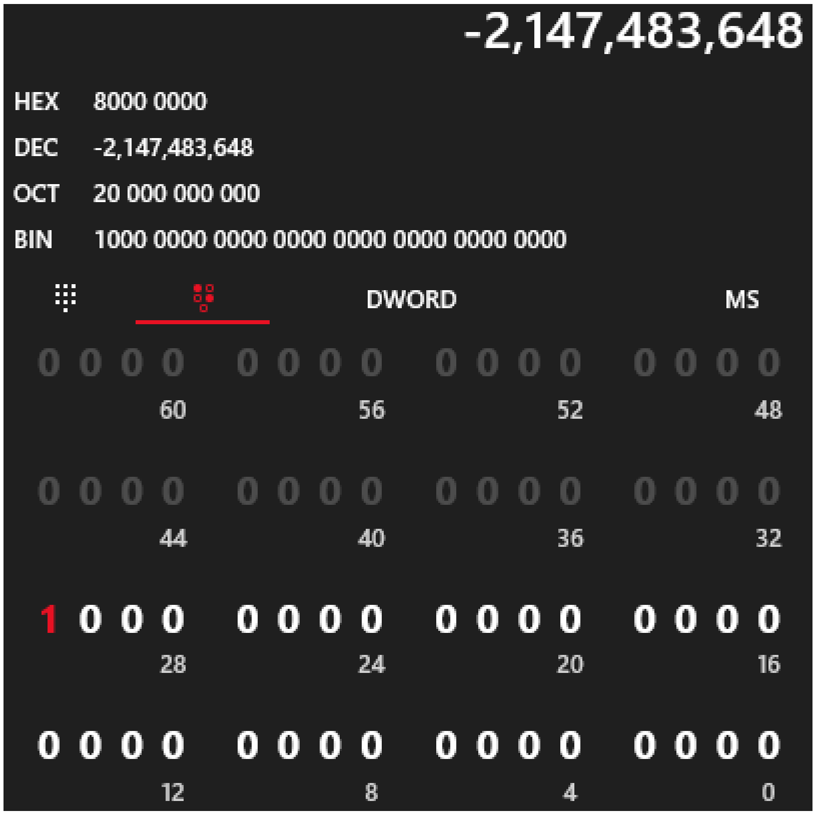
$$-1337 - 1000 = -2337$$
$$-1337 + (-1000) = -2337$$

Now, in order to find a general solution we must figure out which is the negative number that no matter what negative integer it is added to, the result will always be positive (the most significant bit is not set). The solution is:

Decimal: -2147483648
Hex: 0x80000000
Binary: Figure 2.7

It works because the only bit that is set in its binary representation is the most significant one. That means no matter what other negative integer in the range $[-1, -2147483648]$ it is added to since both numbers will have their most significant bit set. The result will be a number with the most significant bit not set, i.e., positive.

You can write -2147483648 (with the minus sign) as the answer for the second level and it will always work. This number is `INT_MIN` since it is the minimum value that can be stored in a `signed int` variable.



```
                                        -2,147,483,648
HEX   8000 0000
DEC   -2,147,483,648
OCT   20 000 000 000
BIN   1000 0000 0000 0000 0000 0000 0000 0000
```

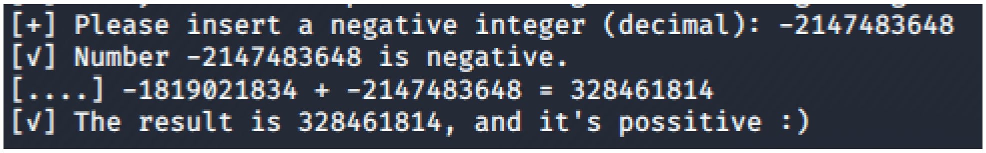
```
0 0 0 0    0 0 0 0    0 0 0 0    0 0 0 0
     60         56         52         48

0 0 0 0    0 0 0 0    0 0 0 0    0 0 0 0
     44         40         36         32

1 0 0 0    0 0 0 0    0 0 0 0    0 0 0 0
     28         24         20         16

0 0 0 0    0 0 0 0    0 0 0 0    0 0 0 0
     12          8          4          0
```

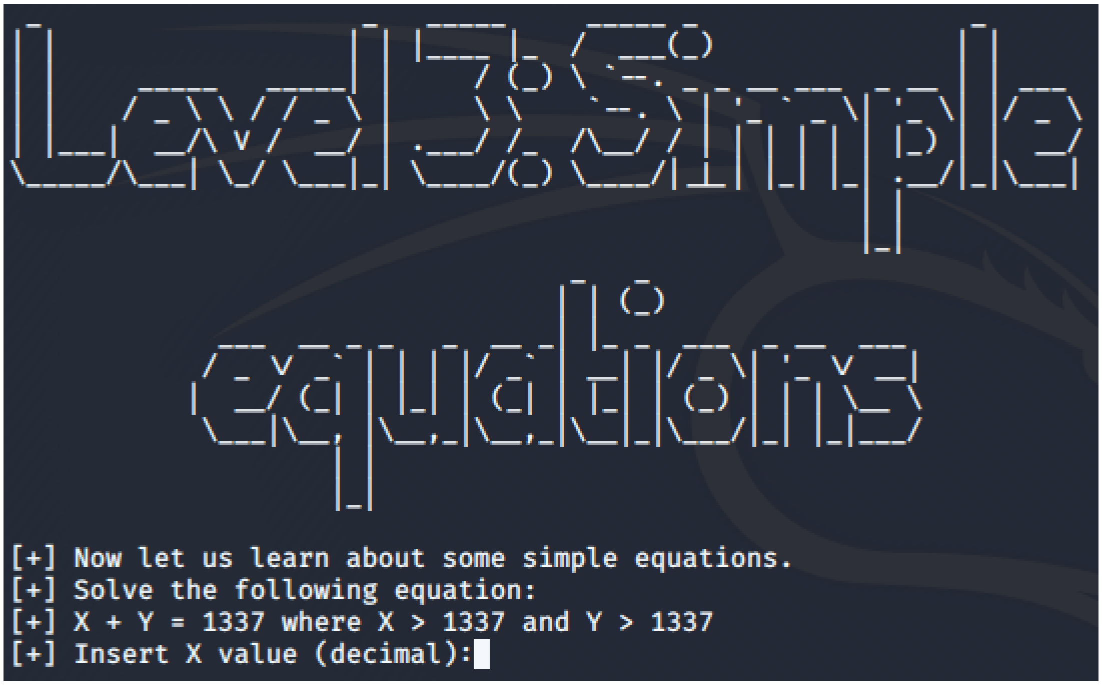
Figure 2.7: General solution to level 2.



```
[+] Please insert a negative integer (decimal): -2147483648
[√] Number -2147483648 is negative.
[....] -1819021834 + -2147483648 = 328461814
[√] The result is 328461814, and it's possitive :)
```

Figure 2.8: Solving the second level.

2.3 Level 3: Simple equations



```
[+] Now let us learn about some simple equations.
[+] Solve the following equation:
[+] X + Y = 1337 where X > 1337 and Y > 1337
[+] Insert X value (decimal):
```

Figure 2.9: Formulation of the third level.

The third level presents a simple equation that must be solved. The equation is:

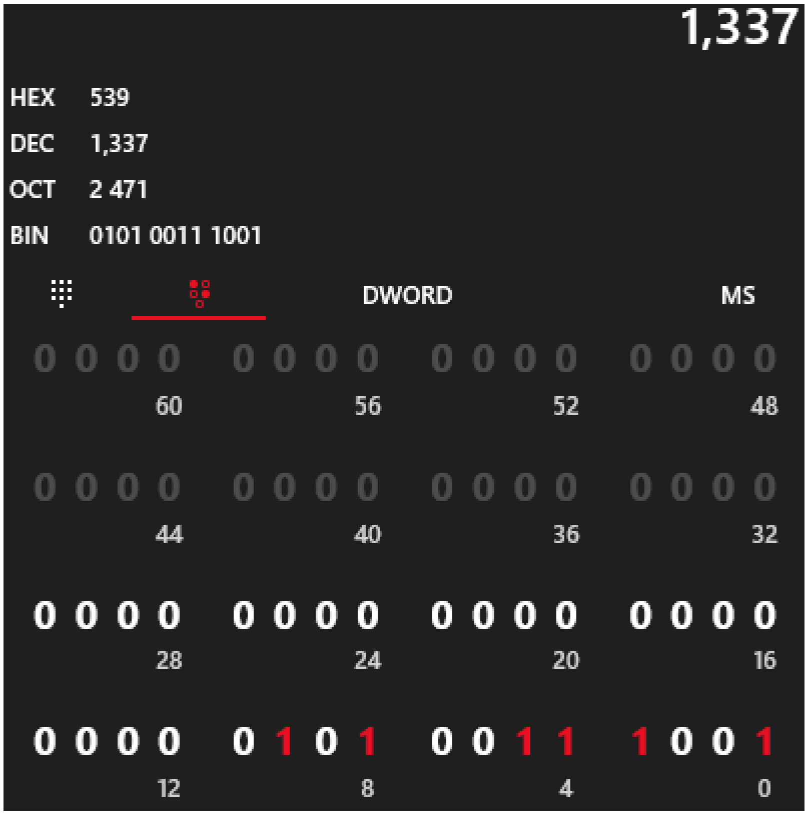
$$\left.\begin{aligned} X &> 1337 \\ Y &> 1337 \\ X + Y &= 1337 \end{aligned}\right\}$$

At a first glance it seems impossible to solve the equation since there is no way $X + Y = 1337$ when both X and Y are greater than 1337. Unsurprisingly enough, there are plenty of alternative solutions to the equation given the context we are working in. Bear in mind we are working with `ints` that are 32 bits long. That is, there will be truncation after the sum.

Given the binary representation of the number 1337, as shown in Figure 2.10, we must find two integers both bigger than 1337 that after adding them up, the first 32 bits of the result are equal to 1337. The reasoning behind it is that only the first 32 bits will prevail in memory, the rest will be discarded (that is what truncation is).

As long as both integers are bigger than 1337 and their addition results in a number whose 32 first bits are also 1337, the solution is valid. That is why there are plenty of solutions to this level. I will show you the easiest solution but it is not the only one.

The solution I used to pass the level consists in preserving the first 32 bit with the value 1337 and simply setting the bit number 33 for the X value, and then unset all bits except the 33rd for the Y value. The logic behind it is truncation based, as mentioned earlier. Since all numbers will be truncated to the first 32 bits because we are working with `ints`.



1,337

HEX 539
DEC 1,337
OCT 2 471
BIN 0101 0011 1001

```
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     44        40        36        32

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     28        24        20        16

0 0 0 0   0 1 0 1   0 0 1 1   1 0 0 1
     12         8         4         0
```

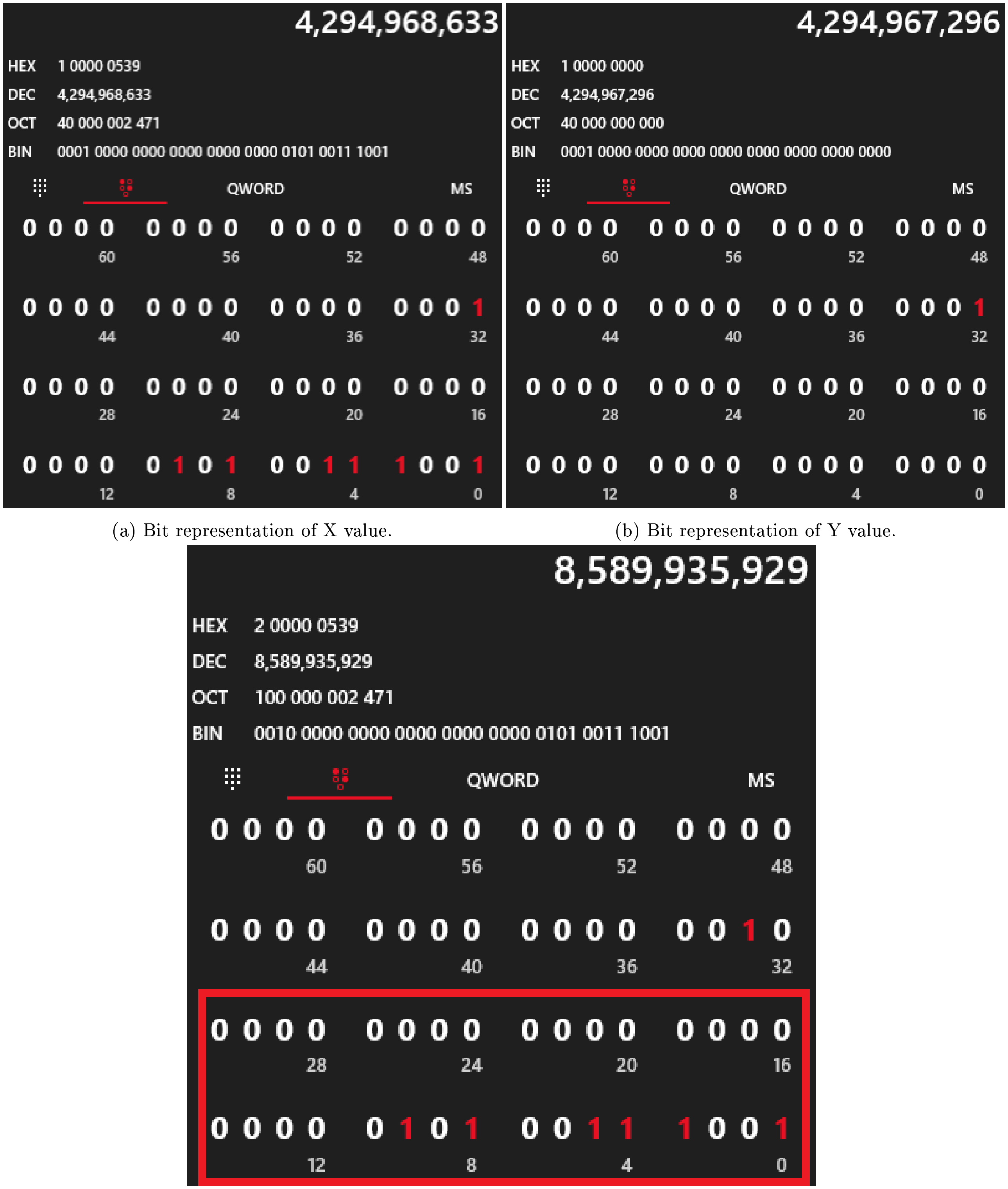
Figure 2.10: Binary representation of 1337.

After inserting 42949686333 value for X and 4294967296 for Y, the level is completed. It works because when a number such as 42949686333 does not fit into a 32 bit long `int` data type, so it gets truncated. The work-flow of the level is the following:

1. The user is asked for input for X variable. User inserts 42949686333. It is bigger than 1337. Program tries to store it into an `int` variable. The number does not fit. It gets truncated to 32 bit. $X = 1337$ after truncation.

2. The user is asked for input for Y variable. User inserts 4294967296. It is bigger than 1337. Program tries to store it into an `int` variable. The number does not fit. It gets truncated to 32 bit. $Y = 0$ after truncation.

3. $X = 1337; Y = 0; X + Y = 1337$. Level passed.



```
4,294,968,633
HEX   1 0000 0539
DEC   4,294,968,633
OCT   40 000 002 471
BIN   0001 0000 0000 0000 0000 0000 0101 0011 1001

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 1
     44        40        36        32
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     28        24        20        16
0 0 0 0   0 1 0 1   0 0 1 1   1 0 0 1
     12         8         4         0
```

(a) Bit representation of X value.

```
4,294,967,296
HEX   1 0000 0000
DEC   4,294,967,296
OCT   40 000 000 000
BIN   0001 0000 0000 0000 0000 0000 0000 0000 0000

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 1
     44        40        36        32
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     28        24        20        16
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     12         8         4         0
```

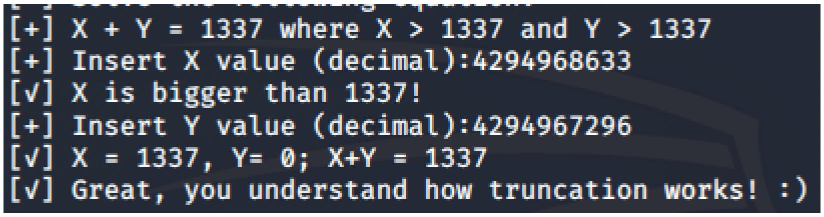
(b) Bit representation of Y value.

```
8,589,935,929
HEX   2 0000 0539
DEC   8,589,935,929
OCT   100 000 002 471
BIN   0010 0000 0000 0000 0000 0000 0101 0011 1001

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 0 1 0
     44        40        36        32
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     28        24        20        16
0 0 0 0   0 1 0 1   0 0 1 1   1 0 0 1
     12         8         4         0
```

(c) Bit representation of the addition of X and Y.

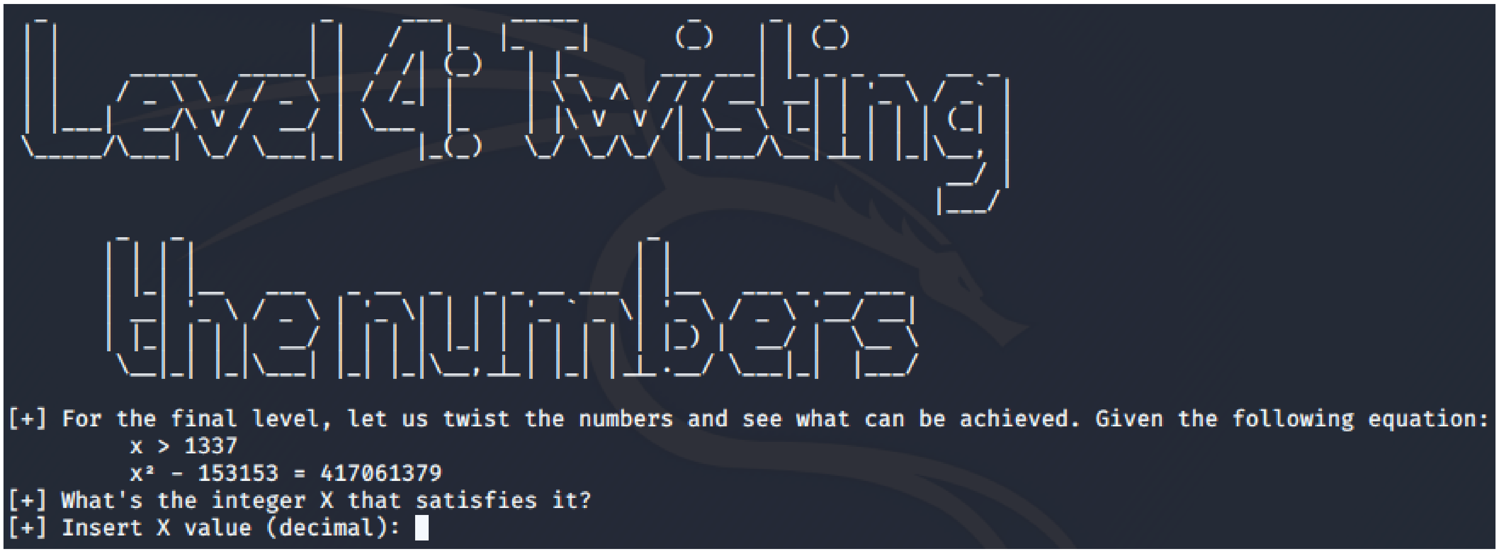
Figure 2.11: Bit representation of one possible solution to level 3. Only the first 32 bits will prevail after truncation.



```
[+] Solve the following equation:
[+] X + Y = 1337 where X > 1337 and Y > 1337
[+] Insert X value (decimal):4294968633
[√] X is bigger than 1337!
[+] Insert Y value (decimal):4294967296
[√] X = 1337, Y= 0; X+Y = 1337
[√] Great, you understand how truncation works! :)
```

Figure 2.12: Solving the third level.

2.4 Level 4: Twisting the numbers



```
[+] For the final level, let us twist the numbers and see what can be achieved. Given the following equation:
        x > 1337
        x² - 153153 = 417061379
[+] What's the integer X that satisfies it?
[+] Insert X value (decimal):
```

Figure 2.13: Formulation of the forth and last level.

The forth and last level asks the user to solve a bit more complicated equation. The equation is:

$$\left.\begin{array}{r} X > 1337 \\ X^2 - 153153 = 417061379 \end{array}\right\}$$

Once again, the provided integer for X must be bigger than 1337. In this case, this condition is irrelevant. This leaves us with the following equation:

$$X^2 - 153153 = 417061379$$

Now, isolating the X variable:

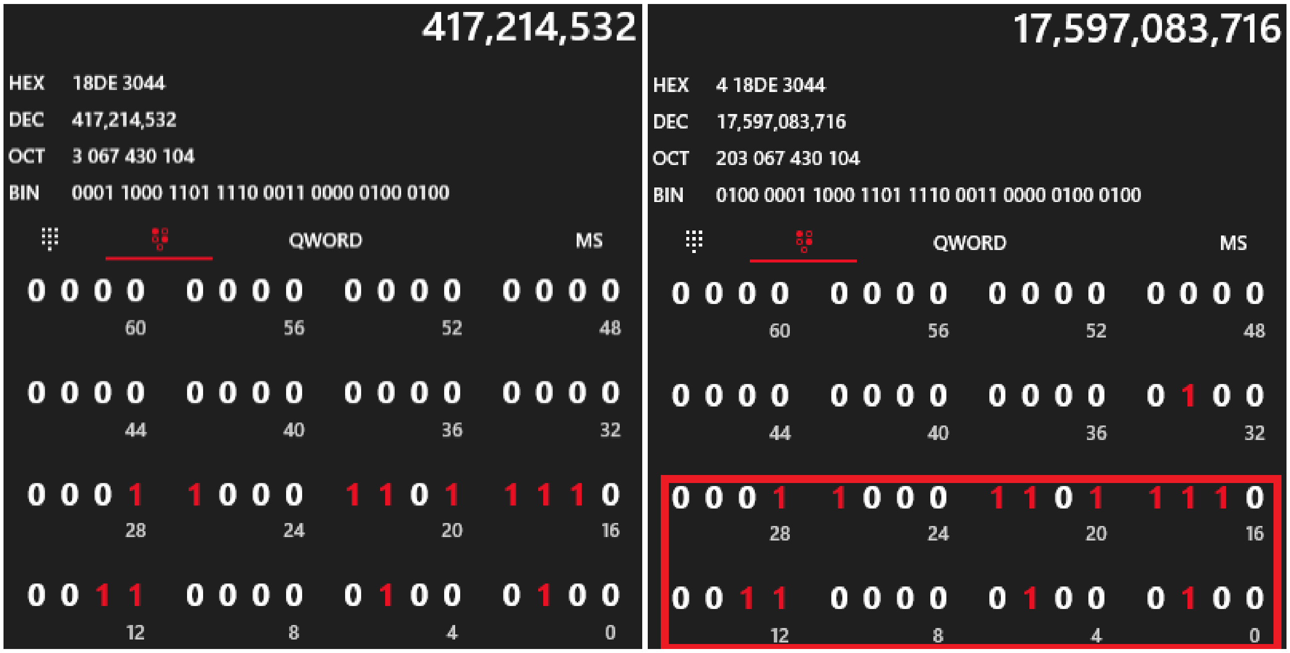
$$X = \sqrt{417214532}; then$$
$$X = 20425.83002$$

Well, this approach is incorrect. Remember we are working with integers, so there are no decimals. While it is true that $20425.83002^2 + 153153 \approx 417061379$, it is not a valid solution. The approach that must be followed is different since, in order to solve the challenge, truncation must be taken into account.

What we know so far is that the program expects an integer that will be squared, subtracted 153153 and then compared with 417061379. That is, there must be some integer X that complies with $X^2 = 417214532$. We have seen earlier that getting the square root of the number is not valid.

Once again, we must think about binary representation of the numbers involved in this operation. The binary representation of 417214532 is shown in Figure 2.14a. What we are looking for is a number that after multiplying it by itself, i.e., squaring it, the first 32 bits of the result represent the number 417214532. In other words, we are looking for a number whose first 32 bits represent the number 417214532 and the result of its square root is a valid integer. The result of the square root is the X we are looking for.

The number that meets the previous conditions is 17597083716 whose representation can be seen in Figure 2.14b. Please remember that we were looking for a number whose square root is a perfect integer and $\sqrt{17597083716} = 132654$. The answer to the level is 132654.



```
417,214,532
HEX   18DE 3044
DEC   417,214,532
OCT   3 067 430 104
BIN   0001 1000 1101 1110 0011 0000 0100 0100

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     44        40        36        32
0 0 0 1   1 0 0 0   1 1 0 1   1 1 1 0
     28        24        20        16
0 0 1 1   0 0 0 0   0 1 0 0   0 1 0 0
     12         8         4         0
```

```
17,597,083,716
HEX   4 18DE 3044
DEC   17,597,083,716
OCT   203 067 430 104
BIN   0100 0001 1000 1101 1110 0011 0000 0100 0100

0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
     60        56        52        48
0 0 0 0   0 0 0 0   0 0 0 0   0 1 0 0
     44        40        36        32
0 0 0 1   1 0 0 0   1 1 0 1   1 1 1 0
     28        24        20        16
0 0 1 1   0 0 0 0   0 1 0 0   0 1 0 0
     12         8         4         0
```

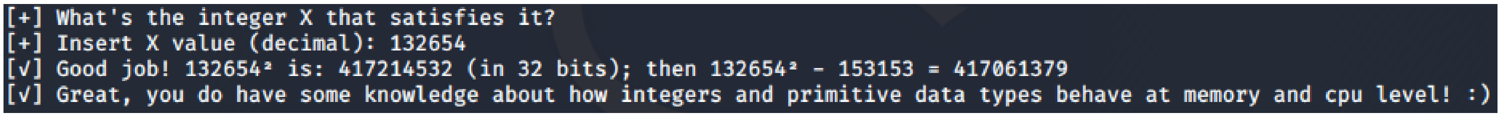
(a) Bit representation of the expected result.

(b) Bit representation of the result of squaring X. Only the first 32 bits will prevail after truncation.

Figure 2.14: Binary representations of numbers involved in level 4.

Notice how 132654 meets level's conditions. It is bigger than 1337. 132654^2 is 17597083716. The number is stored in a `int` variable and gets truncated to 32 bits. 17597083716 truncated is 417214532. This leaves us with 417214532 – 153153 and it indeed is 417061379.

Insert 132654 as the answer to level 4 and now level and whole challenge will be completed.



```
[+] What's the integer X that satisfies it?
[+] Insert X value (decimal): 132654
[√] Good job! 132654² is: 417214532 (in 32 bits); then 132654² - 153153 = 417061379
[√] Great, you do have some knowledge about how integers and primitive data types behave at memory and cpu level! :)
```

Figure 2.15: Solving the forth and last level.

Acknowledgments

A huge thank Ricardo J. Rodriguez, who first taught me about binary exploitation and inspired me to keep learning and go deeper into it.

Bibliography

[1] Blexim, "Basic Integer Overflows .:: Phrack Magazine ::." dec 2002. [Online]. Available: http://phrack.org/issues/60/10.html (Accessed 2020-02-20).

[2] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–29, 2015.

[3] "Integer Overflow - an overview | ScienceDirect Topics." [Online]. Available: https://www.sciencedirect.com/topics/computer-science/integer-overflow (Accessed 2020-02-20).

[4] "Integer overflow - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Integer_overflow (Accessed 2020-02-20).

[5] "CWE - 2019 CWE Top 25 Most Dangerous Software Errors." [Online]. Available: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html#cwe_top_25 (Accessed 2020-02-22).

[6] "In C language, the integer takes 2 bytes for a 32-bit compiler and 4 bytes for a 64-bit compiler. The float always takes 4 bytes. The character always takes 1 byte. So why is there only variation in the case of integers? - Quora." [Online]. Available: https://www.quora.com/In-C-language-the-integer-takes-2-bytes-for-a-32-bit-compiler-and-4-bytes-for-a-64-bit-compiler-The-float-always-takes-4 (Accessed 2020-02-23).

[7] "The New C: Integers in C99, Part 1 | Dr Dobb's." [Online]. Available: https://www.drdobbs.com/the-new-c-integers-in-c99-part-1/184401323 (Accessed 2020-02-23).

[8] "Two's complement - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Two%27s_complement (Accessed 2020-02-23).

[9] R. Lyon, "Two's complement pipeline multipliers," *IEEE Transactions on Communications*, vol. 24, no. 4, pp. 418–425, 1976.