Name(s)____ Period ______ Date _________________

Resource - How and Why Does the Public Key Crypto Work?

How does the *Public Key Crypto Widget* actually work?

Short Version:

Alice's public/private key pair is chosen so that when Alice does `pvt * pub MOD clock` the result is `1.` This means when Alice multiplies Bob's encrypted message by her private key, it effectively cancels out the public key, because it leaves Bob's secret number * 1...which is just Bob's number.

Medium Version:

1. Alice picks a clock size (`clock`) and a private key (`pvt`). Her public key (`pub`) is *specifically* chosen so that `pvt * pub MOD clock = 1`
2. Bob picks a secret value (`secret`) and his public value is computed as: `pub * secret MOD clock` let's call the result `bobPub` . **Important note:** Bob's choice is limited to values strictly less than `clock` (the range: 0 to `clock-1)`
3. When Alice gets `bobPub` she computes: `bobPub * pvt MOD clock`

Now let's look at Alice's final equation, but substitute in the expression for `bobPub`. This gives us:

`(pub * secret MOD clock) * pvt MOD clock`

If you read more below (about how modulo distributes) you know we can refactor this equation to be:

`(pub * pvt MOD clock) * secret MOD clock`

The new underlined portion in the expression is Alice's equation from step 1 -- it works out to `1` because of how we chose Alice's public/private key pair. Thus to finish, we can substitute 1 for that expression, giving us:

`(1) * secret MOD clock` which is just `secret MOD clock`

Because Bob's secret number must be less than clock, we know that `secret MOD clock = secret`

Longer Version with more Details:

Reference: Public Key Crypto Widget Activity

0. **Modulo distributes**
First, a fact about modulo that's important to realize: If you MOD a number once, the result is less than the modulus. And any number that is less than the modulus, when MODed, results in itself. Therefore, if you MOD a number *and then* MOD the result of *that*, you get the same number. For example:

23 MOD 17 = 6
→ 6 MOD 17 = 6
→ 6 MOD 17 = 6…..and so on.

Also, modulo is distributive, which means that if you multiply some numbers and MOD the result, that has the same effect as MODing all the numbers in the first place, then multiply them, then MODing the result of that.

(**27** * **49**) MOD 17 == 14 is equivalent to: [(**27** MOD 17) * (**49** MOD 17)] MOD 17 == 14

The effect of this, as you'll see later on, is that it means as long as MOD is being applied to some string of multiplied numbers, you can MOD any of the individual terms as much as you like. As long as you MOD again at the end, the result will come out the same as if you just did one single MOD operation.

1. **Prime numbers**
The numbers we let you use for modulus (clock size) are all prime numbers. Prime numbers are useful because no numbers divide them, except for 1 and themselves. Modulo is a form of division. So if we only divide values by prime numbers, the results will be unique and distributed evenly over the range of possible values.

2. **Modular multiplicative inverse**
Alice's public key is not random; it is generated based on the public modulus (clock size) and her choice of private key. Alice's public key is calculated in a special way. Here is the equation:

```
(private * public) MOD clock = 1
```

Thinking about Alice's public/private key pair, for example: let's say the public modulus (clock) is 17, and Alice's public key is 5. To figure out her private key X then you have solve this:

```
( X * 5 ) MOD 17 = 1
```

What's interesting about this equation is that you cannot just "solve for x" in a traditional way. The only way to find X is by brute force trial and error - trying out different values for X and plugging them in. You know that X must be in the range 0-16 but beyond that, not much. Furthermore, there is no way to approximate, "get close", or narrow down the answer in any way. You simply have to try out all the values. After some experimentation you'd find that X must be 7, satisfying the equation: (7 * 5) MOD 17 = 1

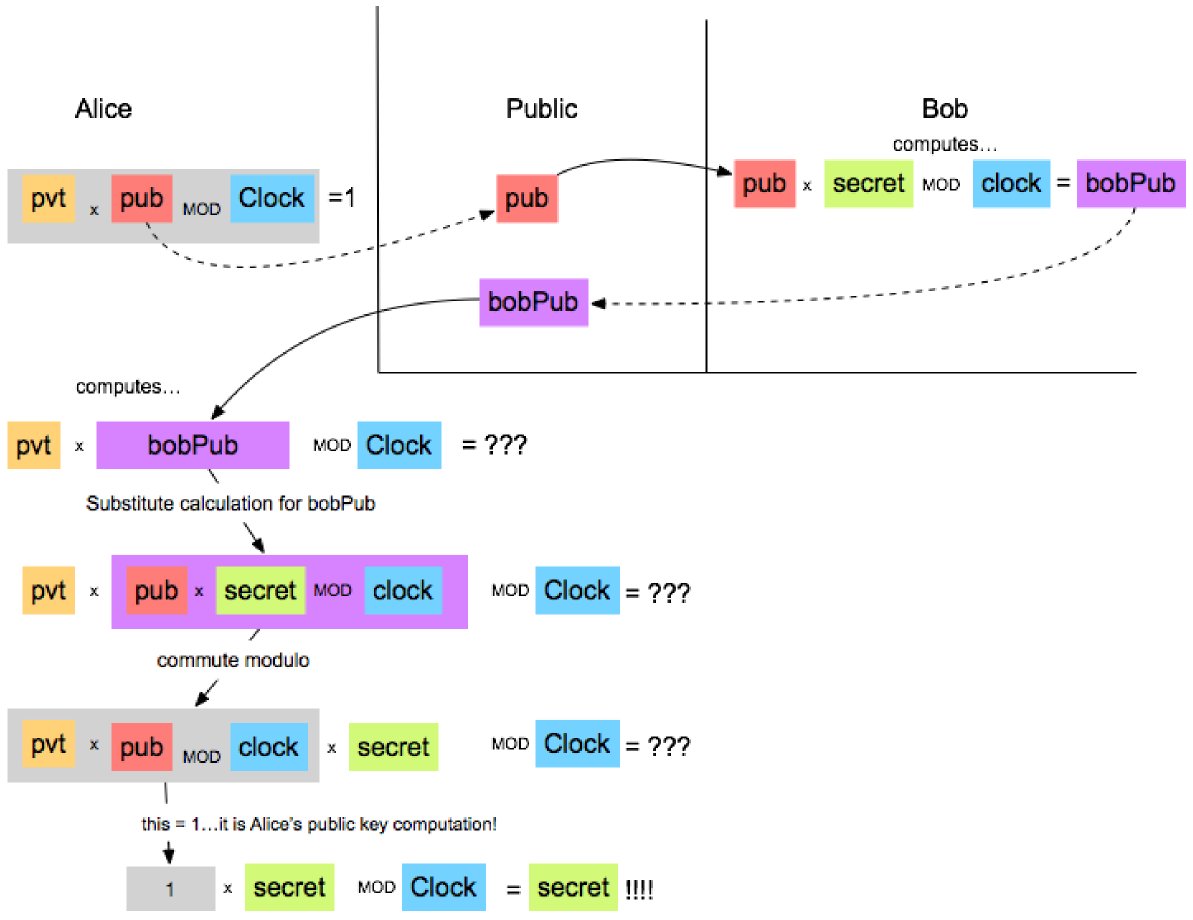
Thus, 5 and 7 form a public/private key pair in our widget (which one is public doesn't matter, they are a reciprocal pair). In the widget we simply calculate ahead of time all of these possible pairs based on whatever is chosen as the public modulus. Bob's secret number, and the resulting public value that he can share, works the same way. In both cases, **the secret value is obscured by the modulo operation, which makes cracking it a more computationally intensive process.**

The math-y term for what we're calculating is the "modular multiplicative inverse".

3. Why Alice can reveal Bob's secret number and Eve can't

Let's now look at the rest of the exchange since it's still not totally obvious *why* Alice can reveal Bob's secret number at the end but Eve cannot. It hinges on the fact that for Alice (public * private) MOD clock **is equal to 1.**

The diagram below traces the exchange of information and tries to reveal what's happening "underneath the hood". Notice the last few steps where we rearrange the values to show how the fact that Alice's public/private key equation is equal to 1 is what allows the secret to be exposed for her. Eve does not have those numbers.



Alice
Public
Bob
pvt x pub MOD Clock =1
pub
computes…
pub x secret MOD clock = bobPub
bobPub
computes…
pvt x bobPub MOD Clock = ???
Substitute calculation for bobPub
pvt x pub x secret MOD clock MOD Clock = ???
commute modulo
pvt x pub MOD clock x secret MOD Clock = ???
this = 1…it is Alice's public key computation!
1 x secret MOD Clock = secret !!!!

Is this *actually* computationally hard for Eve to crack?

The answer is, no, not really. It's hard for a *human* to figure out, because a feature of modular multiplicative inverses is that they are a unique pair for every modulus you could choose AND they are randomly distributed across the number line. So that means there's no heuristic for "getting close"; if you discover a number that gets you one away from your target, you are no closer than if you were 1000 away. So you can't narrow down the field; the only thing you can do is try every possibility. Since the widget maxes out at only 4-digit numbers, you could brute force attack the problem in seconds with a computer program. (In fact, that's what our widget does.) Even for large numbers, it wouldn't be hard to find the multiplicative inverse.

How close is this to the *real* thing?

0. Our scheme here most closely resembles the RSA public key encryption system. (RSA doesn't stand for anything related to cryptography; it's just the initials of the last names of the 3 people who invented it: Rivest, Shamir, and Adleman.)

1. RSA does rely on multiplication and modulo, but instead of just multiplying numbers plainly, the private keys and secret numbers are used as *exponents* when multiplying large numbers, which makes even bigger numbers that are even harder to reverse after modulo.

2. Rather than finding the multiplicative inverse, to crack RSA you need to find the prime factors of a very large number (i.e., given a large number, find two prime numbers that, when multiplied together, yield that number). This is a much harder problem (computationally) than finding the multiplicative inverse (which is what our widget does).

3. The real thing uses VERY large numbers. In this widget, the biggest numbers only had 4 digits (~13-14 bits). The actual numbers used are over 75 digits long (256 bits)! If you think you can fathom how big a number with 75 digits is, you can't. The distance to the edge of our solar system in *inches* is only a 14-digit number.