Lost Modulus

230 points, 42 solves

Statement

```
I lost my modulus. Can you find it for me?
nc 13.112.92.9 21701
crypto-33dee9470e5b5639777f7c50e4c650e3.py
```

Solution

Number theoretic crypto once more. This time we are given some very weird functions:

$A(x) = ((n+1)^x * randint(0, n-1)^n)\ mod\ n^2$

$$B(x) = (\frac{(x^{\varphi(n)}\ \ mod\ \ n^2) - 1}{n} * u)\ mod\ n$$

, where n is a product of two 512 bit prime numbers, $\varphi(n)$ is an Euler's totient function for n (it basically means that $a^{\varphi(n)}\ mod\ n = 1$) and u is a modular inverse of $\varphi(n)$ modulo n. Additionally, the $B(x)$ function call will return only the last byte of the result. We are given a flag encrypted using function A and we can make at most 2048 function calls in order to decrypt it.

After some experimentation one can notice that the B function is in fact a reverse function for A: encrypting message using A and running result through B will return us the original message. Unfortunately, we cannot just simply run the encrypted flag through B, as it will only return the last byte of it. But why exactly is the B function reversing the A-function encryption? Here comes some maths:

First of all we should simplify the A expression. We can notice, that $(n+1)^x\ mod\ n^2 = (n*x+1)\ mod\ n^2$. If you don't believe it you can try to calculate this expression for x equal to 2 or 3 - terms with n^2 and n^3 will disappear after being taken modulo n^2. The full formula can be derived from binomial theorem. So the A function in fact calculates $((n*x+1)*randint(0, n-1)^n)\ mod\ n^2$. This fact will be very important throughout the whole solution.

Now let's try to put the result of simplified A function through B:

$$B(A(x)) = (\frac{(((n*x+1)*randint^n)^{\varphi(n)}\ \ mod\ \ n^2) - 1}{n} * u)\ mod\ n$$

The $randint$ part doesn't look too promising as the random value will distort our message. But worry not! Remember when I said that $a^{\varphi(n)}\ mod\ n = 1$? Our $randint$ in B is raised to the power of $n * \varphi(n)$ and it happens that $\varphi(n^2) = n * \varphi(n)$! So that means that after being taken modulo n^2 it will completely disappear. The $(n*x+1)$ part is also raised to the power of $\varphi(n)$ so it turns into $(n*x*\varphi(n)+1)$. Then we substract 1, divide by n and in the end we multiply by inverse of $\varphi(n)$ to remove the $\varphi(n)$ factor to get the x.

Now we know for sure that B will revert the A encryption. Now it's time for us to decrypt the message somehow, given that the decrypting function only returns the last byte. The very important thing in the B function that we can exploit is returning the value modulo n. Imagine that we have a message at least as large as n. Running it through A and then through B will result in message different from the initial one. We cannot compare the full result of encryption/decryption, although the last byte is enough for us to see if the result has changed (as long as the message is not significantly larger than n, but we know that n is about 1024 bits long, so it shouldn't be a problem for us). It means that we can guess the n value bit by bit starting from the most significant one (let's say 1025th).

Let's start with our variable equal to N. Now using variable i in for loop from 1025 to 0 we will add 2^i to N and check using method described above if our N is larger than n. If not then we will leave N intact. Otherwise we will substract this 2^i from N. This way we can determine the value of n using 2050 queries.

However we can only ask 2048 of them. We can make this recovery process to use less queries although it's kind of a painful job, so I will describe this part at the end. For now, let's imagine that we have around 600 more queries to ask.

Now that we know N we can perform encryption operation ourselves. What's more, given the encrypted message, we can increase its content by k, multiplying it by $(N+1)^k$ (or alternatively - by $N*k+1$). We can also decrease its content by k multiplying it by $1 - N * k$. We will use similar technique we used to find N, to find the flag.

We know the encrypted flag and we can alter its content by multiplying it as I described above. We can find the F value the same way as we found the N value - start with i equal to the message size (we don't know the exact length - I assumed that the message won't be more than 600 bits long). We want to check if our F is larger than flag. Let's multiply the encrypted flag value by $(1 - F * N)$. We can estimate the value of last byte of $flag - F$ as we know the last byte of the $flag$ (and it's equal to 10 - newline character). So now we can see if the $B(encrypted\ flag * (1 - N * F))$ will return the value we expect. If not then it means that $flag - F$ was smaller than 0 and got changed due to being taken modulo N.

So now we know how to find the flag! My method for finding N required around 1500 queries and finding the flag requires 600 queries, so it's slightly too much, but you can remember the result calculated after first run of the program then run it again, starting from smaller bit during flag encoding this time.

Flag: `hitcon{binary__search__and_least_significant_BYTE_oracle_in_paillier!!}`

Slightly better way to find N

This part was very painful. Instead of calling $A(N + 2^i)$ we can call $A(N)$ and $A(2^i)$ then multiply the results and it will give us the same outcome (the *randint* part will change the encrypted message, but the content will be the same). This way, instead of calling A for testing 4 consecutive bits, we can do the following trick: Call $A(current\ N)$ and $A(2^i)$. We know now that $A(N + 2^i) = A(N) * A(2^i)$, $A(N + 2^{i+1}) = A(N) * A(2^i) * A(2^i)$ and so on. We can calculate encrypted values of N we need in order to test the 4 consecutive bits, ourselves. This way instead of 8 calls (2 per bit) we will use 6 (2 * A call + 4 * B call). This changes the number of calls from 2050 to around 1500.

Source code in lostmodulussolve.py (but be careful - it's very ugly).