

Cryptography Systems Series #1

The RSA Cryptosystem Part I: From Theory to Practical Attacks

Gimmick Keys Edition (Version 0.9.9.6)

By Macarthur Inbody

CC-BY-SA

Time Spent Editing:19:24:39 (hh:mm:ss)

Last Updated:06/16/20

Table of Contents

Cryptography Systems Series #1.....	1
Introduction.....	4
Background/The Math behind it.....	5
Creating the Public and Private Keys.....	6
Selecting the numbers for p and q.....	6
Encrypting and Decrypting a Message.....	9
ASCII TABLE.....	9
Encrypting the Message.....	9
Decrypting the Message.....	10
Attacks Against RSA.....	12
e th Root Attack.....	12
e th Root Attack 2 Electric Boogaloo.....	13
Low Exponent Attack via CRT.....	14
Common Modulus Attack.....	17
Starting the Attack.....	17
Factoring the Modulus.....	19
Naive Division Method.....	19
Fermat's Method.....	20
Blind Signing Attack/Signature Forgery.....	22
Starting the Attack.....	22
Appendix 0: Glossary.....	24
Appendix A:Proofs.....	25
Carmichael's Totient proof.....	25
Euler Theorem for Modular Inverse.....	25
e th root attack.....	26
Root attack 2 Electric Boogaloo.....	27
Common Modulus Proof.....	28
Factorization:.....	29
Naive Method.....	29
Fermat's Method.....	30
Hastad Broadcast Attack via Chinese Remainder Theorem.....	31
Appendix B: Algorithms.....	33
Extended Euclidean Algorithm in Python.....	33

RSA Series 1 - Macarthur Inbody CC-BY-SA	3
LCM utilizing the extended Euclidean algorithm.....	34
Modular Multiplicative Inverse.....	35
RSA Bytes to Number Encoder/Decoder.....	37
Common Modulus Attack.....	38
Small Exponent Root Attack.....	39
Fermat's Factors.....	40
Appendix C : Real World Examples.....	42
Radford Factoring/Decryption Challenge.....	42
Radford Common Modulus Attack.....	43
Fermat's Near Prime Attack.....	46
Public Key.....	46
Decoded Vectors.....	46
Cubed Root Attack.....	50
Hastad Broadcast Attack via CRT.....	51
Gimmick Keys.....	59
The Challenge.....	59
Factorization.....	59
Calculating the Decryption Key.....	59

Introduction

This lab will teach you the RSA Cryptosystem and also common attacks against RSA. This may be part of a series. The lab's overall series of events will go as follows. First we will go through the math behind RSA. Then we'll perform some of the common attacks against RSA. If you're interested in the background mathematics that allow the attacks work then see Appendix A. If you want to see the code was used to carry out the attacks then you will have to look at Appendix B. Each algorithm is included in this section for you. The lab will go over the naïve assignment for RSA using ASCII-code assignments. We won't be utilizing the standard encodings for the attacks of this lab so that they can be done by hand.

The code to convert between a number a string of bytes is included in the RSA encoder/decoder section. The techniques in this lab can be applied to any attack no matter how big the numbers are in the end. I hope you enjoy the lab as it will help you carry out attacks when doing CTF Challenges for MECC and also BSidesSWVA and also Radford or any other place where you see challenges that involve RSA.

One last thing I should state until version 1.0 is done this document is not considered complete. I have to still clean up typos and make the flow better. Then lab two will come out which will include Wiener's Attack, and will move the following attacks to that document; Hastad Broadcast, Fermat's Factorization. Further if I can manage to get Coppersmith's various attacks in his paper done they will also be included in Lab two.

Background/The Math behind it.

RSA is based on pure math. The series of mathematical operations that we'll get to later in this lab. For the purpose of this in-class lab we're going to gloss over the proof of how it works that's in the appendix if you'd like to see it.

First you chose two prime numbers p and q that are bit-length x such that when multiplied they make a number that is of bit-length y and that also only has two factors besides 1 and itself that are p and q . Then after multiplying p and q you'll have the modulus n . You will also have to select the public key exponent e and the private key exponent d .

$$\lambda(n) = \text{lcm}(\lambda(p), \lambda(q)) \quad \text{and since } p \text{ and } q \text{ are prime. } \lambda(n) = \text{lcm}(p-1, q-1)$$

To select e , it must satisfy the following constraints.

1. $\gcd(e, \lambda(n)) = 1$

A. That is that $\lambda(n)$ and e share no prime factors.

2. $1 < e < \lambda(n)$

A. That is that e is greater than one and is also less than $\lambda(n)$

B. e must also not be 2 as it will always be divisible by $p-1$, and $q-1$. As p and q must be odd numbers and when you subtract one from them you will get an even number and when multiplied you will also have an even number. Thus, in reality.

3. $3 \leq e < \lambda(n)$

After selecting e , we have to create the private key exponent d that is calculated as follows.

4. $d \equiv e^{-1} \pmod{\lambda(n)}$

A. That is, we are calculating the modular multiplicative inverse of e and $\lambda(n)$

- B. You can calculate this using the extended Euclidean algorithm and the python code to calculate this is given in the Appendix under the section modular_inverse

For this lab we are going to not pad the plain text. Also, we going to do a naive assignment of the message so that you do not have to do the full math. The correct algorithm is talked about once again in the Appendix.

Creating the Public and Private Keys

For this lab you are going to be given the values for the entire encryption and decryption process plus the plain text that we're going to encrypt and decrypt. We will go over the selection process for the private and public key exponents and how to do most of it by hand. Remember that the $\lambda(p, q)$ is equivalent to $\text{lcm}(p-1, q-1)$.

To start off with we are going to just encrypt/decrypt a single byte of data to keep the numbers small.

Selecting the numbers for p and q.

1. We're going to select two numbers for p and q that are both prime and are going to when multiplied give us a value that is at least 3 digits. You need a value of n such that the bit-size of n is large enough to make the chances of a collision impossible. But you also make sure that n does not open you up to the cubed root attack, or coppersmith's attack in the real world. But for this lab we're going to be unconcerned with such issues.
2. We're going to assume that we chose by random chance the values for p and q as given below.
A. $p = 17, q = 7$
3. Now we calculate the modulus n which is done by multiplying p and q.
A. $17 * 7 = 119$
4. Now we have to calculate $\text{lcm}(p-1, q-1) \Rightarrow \text{lcm}(16, 6)$
A. We can do this through prime factorization of both values to do it quickly.

- B. First factor 16 into its prime factors which is. $2*2*2*2$
 - C. Next factor 6 into its prime factors which are $2*3$.
 - D. Next remove the common primes from each value which means that p's primes are now just $2*2*2$, and q is now just 3.
 - E. Then cross multiply the prime factors of p and q. So, $16*3=>48$, and $6*(2^3)=>6*8=>48$.
 - F. Now we know that the lcm between both values is 48.
5. Next we have to calculate the public key exponent e. It has to satisfy the following constraints.
- A. $1 < e < \lambda(n)$. Thus, we can write e as. $1 < e < 48$. Therefore, e must be larger than 1 and also less than 48.
 - B. $\gcd(e, 48) = 1$. Therefore, we have to find a value for e such that it shares no prime factors with the number 48.
 - C. Calculating the factors for 48 we get the following values. $2*2*2*2*3$. We know that e cannot be any of the following factors.
 - 1. 2,3,4,6,8,12,24.
 - D. If we chose a prime number then we all have to do is make sure that it is not a prime factor of $\lambda(n)$. Thus, we know we cannot use 2 or 3. We will go up the prime list until we find one that is larger than 3 but will not go into $\lambda(n)=>48$.
 - E. Further we cannot use 2 as all numbers that are even by definition divisible by 2 and thus the gcd of primes p-1,q-1 will result in an integer n' that is an even as both p and q must be odd numbers to satisfy the rest of the constraints.
 - F. We chose 5 as $\gcd(48,5) = 1$. As they share no primes with each other.
 - 1. Most of the time you'd be choosing a value e that is much larger than this for real messages but we are making the math simpler.
 - G. This value is given to everyone as our public key in the PKI standard.
6. Now we must calculate the private key exponent d. It is calculated via the formula
- $$d \equiv [e]^{-1} \pmod{\lambda(n)}.$$
- A. We are going to use the extended Euclidean algorithm(code shown in the appendix) to calculate the modular multiplicative inverse of e and $\lambda(n)$.

- B. We get the value of 29 by calculating it.
- C. Thus, the private key exponent is 29.
7. The modular multiplicative inverse in this case since e is prime and $\lambda(n)$ is relatively prime can be calculated it with the following formula. This only works if $\gcd(e, \lambda(n)) = 1$. Otherwise you have to do the more complex formula.
- A. $d = e^{(\lambda(n)-1)} \pmod{\lambda(n)}$
- B. $d = 5^{(48-1)} \pmod{48} \Rightarrow 5^{47} \pmod{48}$
- C. $d = 710542735760100185871124267578125 \pmod{48}$
- D. $d = 29$
8. Another way to calculate d is with the following formula. **But** your value for d will be ungodly huge.
- $$d = (1 + n * m) / e \quad n = (p - 1) * (q - 1)$$
- A. $d = (1 + 96 * 119) / 5$
- B. $(1 + 11424) = 11425 \quad 11425 / 5 = 2285$
- C. Thus $d = 2285$.
9. This method is not recommended for real world use because the private key exponent is way larger than it actually has to be (2285 vs 29) but it will work when doing it by hand.

Encrypting and Decrypting a Message

We have the following values for this part of the lab. $n=119$, $e=5$, $d=29$. We are going to encode some ASCII text according to the ASCII table code point for the value.

When reading the table below keep in mind that NUM is the ASCII code point for the character. And CHAR is the printable character.

ASCII TABLE

CHAR	space	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
NUM	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
CHAR	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
NUM	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
CHAR	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
NUM	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
CHAR	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
NUM	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
CHAR	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
NUM	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
CHAR	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
NUM	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	

Encrypting the Message

1. Using the table above we're going to encode the following character. A
 1. Getting its numerical value that makes it 65.
 2. So, $m=65$.
2. Next we're going to encrypt it using the following formula. Where e is the public key exponent, m is our message and n is the modulus.
 1. $c = m^e \pmod{n}$
3. Plugging in the values we get.
 1. $c = 65^5 \pmod{119} \Rightarrow c = 46$
 1. Intermediate values proving it.
 2. $65^5 = 1160290625$
 3. Then $1160290625 \pmod{119} = 46$.
4. Now we have the cipher text 46. To get the plain text we have to decrypt the value.

Decrypting the Message

1. Now we need to decrypt it using the private key exponent d . With the following formula.
 1. $m = c^d \pmod{n}$.
2. Plugging in the values we get.
 1. $m = 46^{29} \pmod{119} \Rightarrow 65$.
 2. Intermediate values.
 1. $46^{29} = 1659499472763109991171612967522797815962278035456$
 2. $1659499472763109991171612967522797815962278035456 \pmod{119} = 65$.
3. Thus, we get out plain text number 65.
4. After converting it back to text we get the letter A.

Attacks Against RSA

e^{th} Root Attack

If the message is small enough then you can calculate the e^{th} root of the cipher-text to get the message back. In the example below we are saying that e is 3. The reasoning behind this is some major math. The primary thing to remember is that if the original M is less than the cubed root of n then we can calculate C as simply being the M^e . Thus all we have to do is take the e^{th} root of C to get back the plain text. This only works with smaller public key exponents though as the operations to find the e^{th} root quickly get out of hand.

1. If we know that $e=3$ and that $M < n^{1/3}$ then we know that $C=M^e$. With this knowledge we can reverse the encryption through the following formula.
 - A. $M = \sqrt[3]{C}$ Since it is the inverse of M^3 . We know that the inverse of raising a value to the power e is simply the e^{th} root of the resulting value.
2. We have the following information.
 - A. $C=729000, e=3, n=1055449$
3. If we plug in the values to the formula above we get the following formula.
 - A. $\sqrt[3]{729000} = M$
 - B. Calculating the cubed root of it we get the value of 90.
 - C. Converting back to ASCII we get the letter Z.
4. Confirming that the values are correct we can encrypt Z again and make sure we get the same cipher text.
 - A. $C = M^e \pmod n$
5. Plugging in our values we get.
 - A. $C = 90^3 \pmod{105449}$
 - B. Solving it we get 729000.
6. Thus, it is proven. We have found out that so long as the message size is less than $n^{1/e}$. We take the e^{th} root of the cipher text so long as the original message block size is less than that value we can use a simple cubed root.

7. The factored values for p and q are given below so that you can calculate d if you really wanted to see the attack in action. The key thing to remember is that in the real(ish) world that the modulus n would be so large that you cannot easily factor it.

A. If we factored n into p and q we would get p=863, and q=1223.

This will also work with other values of so long as they are small enough to deal with and is explained in the Appendix A.

e^{th} Root Attack 2 Electric Boogaloo

We can also generalize the e^{th} root attack to any values of n and C even if the plain-text is P(padding plain text) or any lengths. We can recover P from C using the root attack described above but with some slight modifications. In the next few lines we'll go over how to carry out such an attack. To find the math behind it look into the Appendix as I don't want to repeat myself. You're going to find some number j such that it times the modulus n added to the cipher-text and then taking the e^{th} root of that cipher-text we get back the plain text. One thing to remember is that e has got to be small we're talking less than 11. Also the larger the message the longer it take. In reality it only works for e less than or equal 5 and the message length is less than 3 bytes. It's mostly a toy attack and will be utilized for a CTF chal. Also, the values for p and q should be less than 24bits to keep it(fast enough on my laptop CPU in python). So, we're utilizing 12bit values for p and q. Don't expect this to work for any real RSA keys it's a toy method and only a toy method as it can take a long time. This will only be seen in CTF challenges and you'll know that it is possible if you try all other attacks and nothing else works. That or you hint is to do e^{th} root but sets.

The following values were given to you for this attack.

1. $C=21861$
2. $e=3$
3. $n=63191$

You don't have to worry about n being any certain size as this works for all sizes of n, P, and C. It doesn't matter at all. The example below is utilizing the above values which is the cubed root but it can work for the 5^{th} root also.

1. First you have to setup a formula like so.

A.
$$P = \sqrt[e]{C + (k * n)}$$

2. Plug in the values that we know from before.

$$A. P = \sqrt[3]{21861 + (k * 63191)}$$

3. Now we have to find the value of j. Then after taking that 3rd root of C we'll see if it's an approximate value(as in no decimal portion). If it is then we'll stop because we've found the plain text. If not we'll continue incrementing j until such a time that the value we get for P is a whole number. The first value you get back will be the plain text number.
4. We get j=4. Then we have the following values.

$$A. \sqrt[3]{21861 + (63191 * 4)} \Rightarrow \sqrt[3]{274625} \Rightarrow 65.$$

$$B. \text{ Thus } P = 65.$$

Low Exponent Attack via CRT

Aka Hastad Broadcast Attack

If you have messages that all have the same exponent but different moduli then we can carry out an attack to get the plain text back. You have to have e cipher-texts captured with the same exponent. You have to know the modulus n, their exponent e, and also the cipher-texts. For the attack below we're carrying out the Hastad Broadcast Attack utilizing the Chinese Remainder Theorem to calculate the relationship between all of the cipher-texts. Then we'll be able to calculate the eth root of the common cipher-text value we get to get back the plain text. This is an evolution of the eth root attack if you want to look at it that way. The example below is going to use the value for e as 3 but it works for all values of e but in reality only smaller values of e. If you want to see how the math works go to Appendix B section Hastad Broadcast Attack. This is done in only ~45 steps(by hand), with some code it goes much much faster.

The following values have been captured; e=3.

1. $c_1 = 0x2abd8bd6da91c1$ $n_1 = 0x67a5819556583d$
2. $c_2 = 0x7b55b4c8321fd5$ $n_2 = 0xaea4fc03dc1537$
3. $c_3 = 0x22bd95bcc1d23f$ $n_3 = 0xc9d0e3e3413f33$

Now we need to use the formulas from Appendix B to setup the attack.

1. The Ns

$$A. N = n_1 \cdot n_2 \cdot n_3 \quad N_1 = n_2 \cdot n_3 \quad N_2 = n_1 \cdot n_3 \quad N_3 = n_1 \cdot n_2$$

2. The ds

$$\text{B. } d_1 = [N_1]^{-1} \pmod{n_1}$$

Finally we have to calculate X. We will do this in steps as follows. Both formulas will be shown below.

$$a = c_i \cdot N_i \pmod{N}; \quad b = a \cdot d_i \pmod{N}$$

Then we'll calculate the next cipher-text's values. And use a third variable for it. Then we'll add the value of c(which is the next cipher text's value) to x. Then we'll do it again followed by adding this value again to x. And then we'll calculate $X = x \pmod{N}$ and we finally have the value for X. Then we'll take the eth root of X and we'll have our plain-text value to be decoded. Now it's time for the step-by-step.

$$1. \quad N = (29173898576025661 \cdot 49158048251122999 \cdot 56806147507699507)$$

$$\text{A. } N = 81467509045005285049628743756532685785442810571873$$

$$2. \quad N_1 = 49158048251122999 \cdot 56806147507699507$$

$$\text{A. } N_1 = 2792479340143902858452211788661493$$

$$3. \quad N_2 = 29173898576025661 \cdot 56806147507699507$$

$$\text{A. } N_2 = 1657256785884378298854397109049127$$

$$4. \quad N_3 = 29173898576025661 \cdot 49158048251122999$$

$$\text{A. } N_3 = 1434131913873637995603121491277339$$

For the calculations of d_i I am using the following formula. $d_i \equiv [N_i]^{-1} \pmod{n_i}$ We are also assuming that you've done the calculations. I'll only be showing the first one as you should be able to follow along after that only the values will be shown.

$$5. \quad d_1 \equiv [N_1]^{-1} \pmod{n_1}$$

$$\text{A. } d_1 = 12036145335478373$$

$$6. \quad d_2 = 30682595357216074$$

$$7. \quad d_3 = 54719786082622140$$

Now we are going to calculate X. Once again I'm only going to show the formula in it's generalized form and what we have to do at the end. $x_i = c_i * N_i * d_i$

Then we have to calculate X since $e=3$ we're going to have to have 3 x values. Also for the one that's shown we're showing the numbers hex encoded to keep it all on one line.

$$X = x_1 + x_2 + x_3 \pmod{N}$$

$$1. \quad x_1 = (0x2abd8bd6da91c1 * 0x89ae0b631f355e60849e1607c2f5 * 0x2ac2cf772fa865)$$

$$A. \quad x_1 = 0x3d6eabcb0b1ea18baf5ee4ece6d0be70ebd3eb10b62716baddd6a69$$

$$2. \quad x_2 = 0x10c31879483f80bd199158a3ce1d42f08b59381a3ea23fb972bb103e$$

$$3. \quad x_3 = 0x7496569ed8304461252da56af14398f61d6270caaa03fc37b0da32c$$

$$4. \quad X = (0x3d6eabcb0b1ea18baf5ee4ece6d0be70ebd3eb10b62716baddd6a69 + 0x10c31879483f80bd199158a3ce1d42f08b59381a3ea23fb972bb103e + 0x7496569ed8304461252da56af14398f61d6270caaa03fc37b0da32c) \pmod{0x37be09733ceb163f6774ec3ec0c5a802c7022bac61}$$

$$A. \quad X = 2826123136387388200326536000$$

Next we have to use the formula. $M = \sqrt[e]{X}$

$$5. \quad M = \sqrt[3]{2826123136387388200326536000}$$

$$A. \quad M = 1413829460$$

That is the RSA Encoded number that we have to then decode utilizing the `rsa_ASCII_decode` algorithm in appendix C.

We get the plain-text to be "TEST".

I have already gotten the p and q from the third modulus. Confirming this since I already have p and q which are 232465199 and 244364093 respectively. With $\lambda(n) = 4057581930776444$ and $e=3$ thus $d = 1352527310258815$. Finally we can verify that this attack was correct by encrypting our number with $e=3$ modulus n_3 then confirming that we get c_3 .

$$6. \quad c_3 = 1413829460^3 \pmod{56806147507699507}$$

$$A. \quad c_3 = 9778600022757951$$

- B. Converting to hex that becomes. `0x22bd95bcc1d23f`. Thus we know that we have found the plain-text. The decryption key is there in case you want to do it the hard way but I chose to the fast way.

Common Modulus Attack

If instead different public key exponents are utilized but the same modulus for n is used we can calculate the original plain-text through the common modulus attack. You have to have the same plain-text that's encrypted with different public key exponents but the same modulus values for this attack to work. Once again the math that proves this is in the appendix. We're going to gloss over the math that makes this work like usual for proofs look into the proofs section.

Starting the Attack

You have intercepted two cipher-texts and the public key exponents, and also the modulus for the cipher-texts also.

$c_1=26788046, e_1=29, c_2=53830820, e_2=41, n=110171401$

Seeing as both cipher-texts have the same modulus but different exponents we can carry out a common modulus attack against the cipher-texts to get the plain text.

1. We know that both cipher-texts were created through the following formulae.

A. $c_1 = m^{e_1} \pmod n$

B.

$$c_2 = m^{e_2} \pmod n$$

2. Utilizing the math from below (we're not going to go over the math here as it's far too time consuming.). So, for now we have to solve for the following variables so that we can get the plain-text back.

3. First we have to calculate a $a \equiv [e_1]^{-1} \pmod{e_2}$

A. $a = \text{mod_inv}(29, 41) \Rightarrow a = 17$

4. Then we have to calculate b as such.
 - A. $b = (\gcd(29, 41) - 29 * 17) / 41$
 1. $\gcd(29, 41) = 1$
 2. Remember PEMDAS
 3. $4(1 - 29 * 17) / 41 \Rightarrow (1 - (29 * 17)) / 41 \Rightarrow (1 - 493) / 41 \Rightarrow -492 / 41 \Rightarrow -12$
 - B. $b = -12$
5. Next you have to calculate i such that. $i \equiv [c_2]^{-1} \pmod{n}$
 - A. $i = \text{mod_inv}(c_2, n)$
 - B. $i = \text{mod_inv}(53830820, 110171401)$
 - C. $i = 32332591$
6. Finally, we have to calculate both m_x and m_y .
7. First m_x .
 - A. $m_x = [c_2]^a \pmod{n}$
 - B. $m_x = 53830820^{17} \pmod{110171401}$
 - C. $m_x = 37473290$
8. Then m_y .
 - A. $m_y = i^{-b} \pmod{n}$
 - B. $m_y = 32332591^{-(-12)} \pmod{110171401}$
 - C. $m_y = 74045807$
9. Then finally m.
 - A. $m = m_x * m_y \pmod{n}$
 - B. $m = (37473290 * 74045807) \pmod{110171401}$
 - C. $m = 828365$
10. Now we use the naive ASCII encoding to decode the code-points.
11. Message is RSA.

If you'd like to see the math behind it look at the appendix.

Factoring the Modulus

YES REALLY.

Two “simple” methods.

Naive Division Method

The first method is the naive blind division method utilizing elementary number theory. For the rest of this lab document when I say factors I am excluding 1 and the number N. Let's say we want to factor a number N that has only 2 distinct prime factors that we'll call a and b. Using basic number theory we know that $a \vee b < \sqrt{N}$. Therefore, all we have to do is take the square root of the number N and then start counting down from that value and trying every prime that is possible until we find one that divides evenly without a remainder. Once we've found this number then we know that the product of the division is b. We can prove this through the following simplified example.

Assume that $N=91$. We need to factor N and get the two prime products a and b.

1. First we have to calculate $i = \sqrt{N}$. Plugging into it the value we have for N we can calculate i.
 - A) $i = \sqrt{91}$ therefore $i = 9.5$ rounding it down we get. As it has to be less than that value.

$$i \simeq 9$$
2. Next we can use the following knowledge to try values in the following range.
 - A) $3 \leq a < i$
 - B) We know that a has to be greater than or equal 3 as 3 is the smallest possible prime. Also, a has to be less than or equal to i since $i < \sqrt{N}$.
 - C) With this knowledge we'll have to start at i and start counting down and trying each prime or we can count up from i. If we count up from i then it has to satisfy the following constraints.
 - I. $i < a < N$
 - D) For this lab we're going to be counting down from i.

E) It can be done with the following constraints.

I. Try N/i while $i \geq 3$

1. reduce i if N/i returns a remainder.

II. Then when no remainder is found after division then the product is b .

III. This can be expressed as follows.

1. While $N \bmod i \neq 0$:

1. $i=i-1$

2. $b= N \bmod i$

3. $a=i$

F) Using our math above the next prime below 9 is 7. $91/7=13$

G) Thus, we have found the two products and they are 7 and 13 as $13*7 = 91$.

Fermat's Method

This is slower than trivial division unless the primes are close to each other. And for this exact lab you're in luck they are close. We can factor the primes using Fermat's method with the following method. You'll be finding the prime factors p and q again using some elementary number theory. You set x to the floored value of the square root of n . Then you'll set b as $a^2 - n$. While the square root of b is not a whole number then add one to a and then repeat the process until the square root of b is a whole number. The floor value is when you round any number down to the next whole integer.

1. Set $a = \text{floor}(\sqrt{n})$

2. Set $b = (a^2) - n$

3. while $\sqrt{b} \neq \text{floor}(\sqrt{b})$:

1. $a = a + 1$

2. $b = (a^2) - n$

4. Then when the loop is broken, you calculate the prime factors p and q as follows.

5. $p = a + \sqrt{b}$

6. $q = a - \sqrt{b}$

7. Then you have the primes p and q .

Now we're going to do a similar factorization but with a new value of n so that it's faster. $N=29177$.

1. $a = \text{floor}(\sqrt{29177}) \Rightarrow a = 170$
2. $b = (170^2) - 29177 \Rightarrow b = -277$
3. $\sqrt{-277} \Rightarrow 16.6 \cdot \text{floor}(16.6) = 16 \cdot 16 < 16.6$ thus, we increment a and try again.
4. $a = 170 + 1 \Rightarrow a = 171$
5. $b = (171^2) - 29177 \Rightarrow b = 8$
6. Since $\text{floor}(8) = 8$. We now have the value for a and b.
7. $p = 171 + 8 \Rightarrow q = 179$
8. $q = 171 - 8 \Rightarrow q = 163$
9. Confirming that $p \cdot q = n$.
 1. $179 \cdot 163 = 29177$

We have factored n using Fermat's method. This is only useful when the primes are near each other, otherwise the time to factor the primes would take way too long. And by near each other I mean the rules below.

1. p and q are in some set of primes Z . $p, q \in \text{primes } Z$
2. The index i represents the position in the set Z of all primes. And k is small. $p = Z_i$ $q = Z_i + k$
3. Generally this attack will work if p and q both share at least half of their upper bits. That means that if the numbers are 16 bits in length (really small but good for this example).
 - A. $p = 11111111111110001$ $q = 11111111100000111$
 1. $q = 65521$ $q = 65287$
 - B. Since as we can see p and q share the top 8 bits then this attack will work.
 - C. With the above numbers. Fermat's factorization takes just 1 iteration whereas a naive division takes 11 iterations.

This method works for primes of any size even 2048 prime numbers which means you have well over 600 digits in the value of n. You would utilize this attack whenever you see the following clues "Close Primes", "Fermat", "Near Primes", "Reduced Set" or something similar. An optimized version of Fermat's

factorization method using a sieve is actually faster than trivial division but that is left as an exercise to the reader.

Blind Signing Attack/Signature Forgery

For this attack we are going to forge a signature on a message. Now for good reasons Alice may not want to sign Bob's message so he adds some random integer r and combines that with his message he wants signed. Then Alice signs this message and Bob can remove the integer r to get a signature on his original message.

First assume the following information. Also assume that you do not know the private key exponent d but I will be showing it here to show the attack in action.

The following condition must be met for this attack to work.

1. r must be co-prime with n
 - A. This means that $\gcd(r,n) = 1$
2. m should be small enough such that n will be large enough to be reversible.
 - A. This won't matter in the real world though.

$N = 0xac8d218afd60059893df97$

$e = 0xd0ff$

$m = \text{"Don't sign."}$

$r = 163$

d is secret but I'm showing it here so that you know that it works.

$d = 0x10c7f26effb06d61d9614f$

Starting the Attack

1. First encode the message m into M
 - A. $M = \text{rsa_ascii_encode}(m, \text{len}(m))$
 - B. $M = 0x446f6e2774207369676e2e$
2. Use the following formula to calculate M' and plug in our values $M' = M \cdot r^e \pmod n$

A. $M' = 0x446f6e2774207369676e2e \cdot r^{0xd0ff} \pmod{0xac8d218afd60059893df97}$

B. $M' = 0xa6570c30f9f01ea0ccca1b$

3. Next we have to get Alice to sign our Message M'

A. Recall that signing a message is calculated by the following formula. $S = M^d \pmod{n}$ Or basically the same formula as encryption except we're applying it to the plain text M .

B.

$$S = 0xa6570c30f9f01ea0ccca1b^{0x10c7f26effb06d61d9614f} \pmod{0xac8d218afd60059893df97}$$

I. Since we were encrypting M' we're going to call S S' this time.

II. $S' = 0x9507042faaa13f1e403515$

4. Now we need to convert this signature into the signature that we want. This is done through the following formula. $S = S' \cdot r' \pmod{n}$ where $r' = [r]^{-1} \equiv 1 \pmod{n}$

I. $r' = 0xa96020ecf26df5c999012b$

A. Now insert it into the formula.

B. $S = "0x9507042faaa13f1e403515" * "0xa96020ecf26df5c999012b" \pmod{0xac8d218afd60059893df97}$

I. $S = 0x3e1b3b361c5d21073ca47a$

5. Thus we now have the signature $0x3e1b3b361c5d21073ca47a$ on our original message M .

6. Now to confirm that this is the same signature as if we had had Alice sign our message M .

A.

$$S = 0x446f6e2774207369676e2e^{0x10c7f26effb06d61d9614f} \pmod{0xac8d218afd60059893df97}$$

B. $S = 0x3e1b3b361c5d21073ca47a$

7. We have confirmed that our fake signature is the same as the real signature thus the attack is complete.

Now the attack is complete and we have managed to forge the signature. Now in the real world this won't work as most Signing systems will sign a checksum of the message thus you cannot as easily remove the blinding factor.

Appendix 0: Glossary

Here is where I'll list all terms throughout the text that you will need to know so that it is not repeated dozens of times.

$\text{mod_inv}(a,b)$ =modular multiplicative inverse of a and b.

λ = Lambda - Carmichael's Totient function.

Φ = Capital Phi - in this paper it means any Totient function but in the real world it's Euler's Totient function.

\equiv = Equivalent to. Used when mapping congruences.

ϕ = Lowercase Phi - Euler's Totient Function

lcm =Least Common Multiple

gcd =Greatest Common Divisor

floor =rounding down to next whole integer. Also this form is seen. $\lfloor 7.3 \rfloor \Rightarrow 7$

ceil =rounding up to next whole integer. e.g. $\lceil 7.3 \rceil \Rightarrow 8$

All modular multiplicative inverses are written with the following formula. $\text{inv} \equiv [a]^{-1} (\text{mod } b)$ in the proofs. Basically the value we are looking for is on the left-hand side. And it shows $\text{mod_inv}(a,b)$

$\in \mathbb{Z}$ = means in the set of Z. Where Z is all integers.

$\sqrt[e]{I}$ = the eth root of I.

\sqrt{I} = the square root of I.

$a \vee b$ = either a or b.

$a < b$ = a is less than b.

$a > b$ = a greater than b

$a \leq b$ = a less than or equal to b

$a \geq b$ = a greater than or equal to b

$a ** b = a^b$ means a raised to the power b. This is done with the value b is too large to fit into a formula.

Appendix A:Proofs

AKA: There's too many darn letters in my formulae.

Carmichael's Totient proof.

Since the original standard used $\varphi(n)=(p-1)(q-1)$. Since $n=p \cdot q$, and $\lambda(n)=lcm(\lambda(p), \lambda(q))$, and since p and q are prime. $\lambda(p)=\varphi(p)=p-1$ and likewise $\lambda(q)=\varphi(q)=q-1$. Therefore $\lambda(n)=lcm(p-1, q-1)$. To calculate $\lambda(n)$ one must simply use the following formula. Since we know that the following formula can be utilized to calculate the lcm of two values.

$$lcm(a, b) = \frac{a * b}{gcd(a, b)}$$

Thus, we can calculate $\lambda(n)$ with the following formula.

$$\lambda(n) = \frac{(p-1)(q-1)}{gcd(p-1, q-1)}$$

If instead you have a value n that is made up of the same prime some number of times then you can use this formula $\lambda(n)=lcm(p-1, q)$. Where p is prime and q is that prime multiplied by itself some number of times. This value is then able to be utilized by the modular inverse function using the extended euclidean algorithm as euler's theorem is not possible because $\lambda(n)$ is not co-prime with the value n.

Euler Theorem for Modular Inverse

If $gcd(a, m)=1$ then we can use Euler's theorem. The ϕ character represents a Totient function below.

$a^{\phi(m)-1} \equiv 1 \pmod{m}$ Therefore $a^{\phi(m)-1} \equiv a^{-1} \pmod{m}$ Finally we can solve the modular multiplicative inverse through the following formula.

$$x = a^{\phi(m)-1} \pmod{m}$$

And since we know due to the proof above that if m is prime then $\phi(m) = \lambda(m) = m - 1$ holds true. Then we can simply input into the formula thusly $x = a^{m-2} \pmod{m}$. But if instead m is not prime then we'd have to first factor m and then use the formula above to calculate $\text{lcm}(x_i, x_j \dots x_l)$ where each x is an index of all of the prime factors of m . We would find the least common multiplier between all factors (first) then we'd subtract one from each factor and find the lcm between them all.

E.g. $a=16, m=273$

1. For this example we're going to use Carmichael's Totient on the number. That is why there is factoring here. It is simply to make the math easier.
2. Factor 273 into it's primes.
3. $2+7+3=12$. Divisible by 3. So 3 is one prime. Now remainder is 91.
4. After finding the next prime it can be divided by
 - A. We get 7. as $9+1=10$ which isn't divisible by 3, and it doesn't end in a zero of a five thus 5 is out of the question and the next prime is 7.
5. $91/7=13$. 13 is remainder.
6. 13 is the final prime factor.
7. Calculate $\text{lcm}(13-1, 3-1, 7-1)$ Or $\text{lcm}(12, 2, 6)$ which is 12 as 12 is even and the rest of the numbers are factors of 12.
8. Thus $\lambda(m) = 12$.

Plug it into the formula.

1. $x = 16^{(12-1)} \pmod{273}$
2. $x = 17592186044416 \pmod{273}$
3. $x = 256$

Note though that by default since you're already going to be calculating the greatest common divisor between a and m in your code it's simpler to just use the extended euclidean algorithm and bezout's coefficients if you're doing it in code. But for a by-hand method this is preferred as it's much simpler to do.

eth root attack

The e^{th} root attack works so long as e is small enough such that the following constraints are met.

$$1. \quad M = \sqrt[e]{C} \quad \text{if } M^e < n$$

Then we'd know that the following formula works for calculating M from C.

$$M = \sqrt[e]{C}$$

If $M < n^{1/3}$ then $M = \sqrt[3]{C}$ which is another way of evaluating the formula we can also write the formula as follows. $M = \sqrt[e]{C} \quad M < n^{1/e}$

Example $n=11095304447$, $M=90$, $e=5$.

1. $90^5 = 5904900000$
2. $\sqrt[5]{11095304447} \approx 102$
3. Since $90^5 < 11095304447$ and $90 < \sqrt[5]{11095304447}$
4. We know that we can take the 5th root of 5904900000 and get the proper value.

Root attack 2 Electric Boogaloo

There is another type of attack that can be performed so long as e is also small. If $M^e \geq n$ then we can setup a formula where we find the original plain-text through the following formula.

$$P = \sqrt[e]{C + kn} \quad \text{where } k \in \mathbb{Z}$$

Then we can add multiples of n and try each of them until we find the original plain text without having to factor the relatively large n .

$$P = \sqrt[5]{128963428 + (k \cdot 206283449)}$$

After using the algorithm in Appendix B, we get the final value back for the plain-text.

$$P = \sqrt[5]{128963428 + (28 \cdot 206283449)} \Rightarrow P = \sqrt[5]{5698616551} \Rightarrow P = 90$$

Proof

$$90^5 \bmod 206283449 = 128963428$$

Common Modulus Proof

RSA works through the following formula. $C = M^e \pmod{n}$. Assuming that we can intercept two messages that have the same modulus but just different public key exponents defined as follows.

$$c_1 = m_1^e \pmod{n}$$

$$c_2 = m_2^e \pmod{n}$$

Then utilizing bezout's theorem that states if there are integers a and b that are both not zero then there exists integers x and y that $xa + yb = \gcd(a, b)$. Then to solve for x we can utilize the following formula. $x \equiv a^{-1} \pmod{n}$ which is the modular multiplicative inverse of a and n. Finally, we have to make sure that the $\gcd(e_1, e_2) = 1$ so that there is a modular multiplicative inverse to solve for a and b.

$$\text{Therefore } C_1^x * (C_2^{-1})^{-y} = (M_1^e)^x \cdot (M_2^e)^{y1}.$$

The plain text can be represented as simply

$$(c_1^a) + (c_2^b) = m$$

If we insert our values of a and b into the original equations then we will get the following values.

$$m^{(e_1 \cdot a + e_2 \cdot b)} \Rightarrow m^1 \Rightarrow m$$

Finally, with the previous knowledge we can calculate the plain-text with the following formulae. One issue we have to deal with is if b is negative with most of the time it is. Thus, we have to calculate an intermediate value for i, then we have to plug it into a formula also.

We have to find the value i such that. $i^{-b} = c_2^b$.

$$1. \quad a \equiv [e_1]^{-1} \pmod{e_2}$$

$$A. \quad a = \text{mod_inv}(e_1, e_2)$$

$$2. \quad b = (\text{gcd}(e_1, e_2) - e_1 \cdot a) / e_2$$

$$3. \quad i \equiv [c_2]^{-1} \pmod{n}$$

$$A. \quad i = \text{mod_inv}(c_2, n)$$

$$4. \quad m_x = [c_2]^a \pmod{n}$$

$$5. \quad m_y = i^{-b} \pmod{n}$$

$$6. \quad m = m_x \cdot m_y \pmod{n}$$

Thus, we have recovered the plain-text m . Once again the code that implements this is included in the appendix B under “Common Modulus Attack”.

Factorization: Elementary Number Theory

Naive Method

If we are trying to factor a number n that we know is a composite number then we know that at least one of its factors will be $a \leq \lfloor \sqrt{n} \rfloor$. We can extend this knowledge to the composite numbers that exist within RSA as it only has 2 factors besides 1 and itself, the primes p and q . Thus and with this knowledge can start factoring n by trying every number that is less than the $\text{sqrt}(n)$. If we have a prime sieve then we can try each prime less than the square root of n .

We can do this by either

1. Counting up every prime in the set $3 < p < \lfloor \sqrt{n} \rfloor$

2. Counting up from for every prime $N > p > \lfloor \sqrt{n} \rfloor$

We know this will work because for any value $a^2 \leq n$ for any composite number. For our factoring methods we know that $p^2 < n$ because there is a second prime that has to be larger than 3 and p^2 cannot be n or otherwise there would only be one factor and you could easily factor the number by calculating the square root of n . This method is the naive method of factoring n .

Big O complexity is $O(\sqrt{n})$ for this algorithm.

Fermat's Method

If we utilize Fermat's Division method we can factor the values even faster. You can calculate a as $a^2 - n = b^2$. First you find the square root of the number n then round to the next nearest integer. We try all possible values until we find a value of a that is square. More generally.

The algorithm goes as follows.

1. Try $a = \lfloor \sqrt{n} \rfloor$ set $b = a^2 - n$
2. Take \sqrt{b} ,
3. if $\lfloor \sqrt{b} \rfloor^2 \neq b$ then increment a and try again.
4. Once $\lfloor \sqrt{b} \rfloor^2 = b$.
 - A. Set $p = a - b$
 - B. Set $q = a + b$
5. You now have the prime factors of n .

This method is slow though taking $O(\sqrt{n})$ iterations at worst to complete, but if the primes are close to

each other then it will take $O\left(\frac{\Delta^2}{4n^{(1/2)}}\right)$ where $\Delta = |p - q|$ (formula from B. D. Weger 2002). The

variables p and q are the two primes that make up n . This method can be improved through the use of a sieve

to factor larger numbers but still won't be the best possible case for factorization. The source code for this method is in the Appendix B under `fermats_factors`.

There is a sieve improvement to the algorithm but I'm not including that for now.

Hastad Broadcast Attack via Chinese Remainder Theorem

The Chinese remainder theorem states that the map maps all congruences modulo N to a set of n_i . There are linear congruences $x \equiv c_i \pmod{n_i}$, $x \equiv c_j \pmod{n_j} \dots x \equiv c_z \pmod{n_z}$ has a solution that is unique for $n_i, n_j \dots n_z$. If $n_i, n_j \dots n_z$ are co-prime. Or more simply $\gcd(n_i, n_j) = 1$ for all values where $i \neq j$. If we are going to solve for the first two moduli then the formula would be.

$$x \equiv a_{1,2} \pmod{n_1 n_2}.$$

If we utilizing the extended euclidean algorithm to calculate bezout's coefficients then we can utilize it to solve for these inverses as such. $X = a_1 m_2 n_2 + a_2 m_1 n_1$ where m_1 and m_2 are Bezout's coefficients of the values n_1 and n_2 which can be calculated from the `gcd_fast` function in the Appendix B. Now we're going to write the formula's using the simplified formula.

$$\text{The set of } N = n_1, n_2, n_x \quad x \equiv c_1 N_1 d_1 + c_2 N_2 d_2 \dots c_z N_z d_z \pmod{N} \quad \text{where the } N_i = N/n_i$$

$$d \equiv N_i^{-1} \pmod{n_i}$$

If you're public key exponent is 3 then we'd setup the formulas(each one calculated separately) for them with 3 cipher-texts, and 3 moduli. Assuming that we're making each cipher-text is numbered. We'll create some cipher text first. And will setup each section of the formula separately.

Here's the overall formula $X \equiv c_1 N_1 d_1 + c_2 N_2 d_2 + c_3 N_3 d_3 \pmod{N}$ but for sake of simplicity we'll setup each of the parts separately. Below you'll see the formulas for the 3 values. For the attack see the write up. All of the formulas below are denoting the modular inverse with the standard notation not how they were calculated which means they're shown as $x \equiv [a^{-1}] \pmod{b}$ After calculating X use the following formula get back the original plain-text M . $M = \sqrt[3]{X}$ If you were utilizing a different value

for the public key exponent then it'd be similar to the e^{th} root attack where you'd take the e^{th} root of the final value for X to recover the plain text. You'd have to also capture e cipher-texts, and moduli.

1. $x_1 = c_1 \cdot N_1 \cdot d_1$ where $d_1 = [N_1]^{-1} \pmod{n_1}$ and $N_1 = N/n_1$
2. $x_2 = c_2 \cdot N_2 \cdot d_2$ where $d_2 = [N_2]^{-1} \pmod{n_2}$ and $N_2 = N/n_2$
3. $x_3 = c_3 \cdot N_3 \cdot d_3$ where $d_3 = [N_3]^{-1} \pmod{n_3}$ and $N_3 = N/n_3$
4. Finally add each section together modulus N
 - A. $X = x_1 + x_2 + x_3 \pmod{N}$
5. And we get back x, then we take the cubed root of X.
6. $M = \sqrt[3]{X}$

If you want to see it in action refer back to the section where the attack was actually carried out, or the Appendix C "Real World Attacks" Subsection "Low Exponent - Hastad Broadcast Attack".

Appendix B: Algorithms

Note all python code is tab/space sensitive. All code is using a monospaced font to make it easier for you to see the code. I am utilizing 4 spaces for the indentation level so keep that in mind when you utilize this code yourself. All code is licensed under the AGPLv3 or Later. This code is simply here if you want to see how the sausage is made. They are all good functions to include in your CTF arsenal and will give you a major leg up on the competition.

Extended Euclidean Algorithm in Python

Non recursive version. Returns a tuple of gcd, bezout coefficients x and y.

#python supports || assignments thus we don't need a temporary variable to hold intermediate values.

```
def xgcd(a, b):
    x0, x1, y0, y1 = 0, 1, 1, 0
    while a != 0:
        q, b, a = b // a, a, b % a
        y0, y1 = y1, y0 - q * y1
        x0, x1 = x1, x0 - q * x1
    return b, x0, y0
```

Recursive version. Does the same as above but with recursion.

```
'''
```

gcd calculator using the Generalized Extended Euclidean Algorithm.

Python implementation of the extended euclidean algorithm for calculating the gcd.

This code is the recursive variant as it is simpler.

```
'''
def gcd_fast(a,b):
    gcd=0;
    x=0;
    y=0;
    x=0
    #if a or b is zero return the other value and the coeffecient's
    accordingly.
    if a==0:
        return (b,0,1)
    elif b==0:
        return (a,0,1)
    #otherwise actually perform the calculation.
    else:
        #set the gcd x and y according to the outputs of the function.
        # a is b (mod) a. b is just a.
        gcd, x, y = gcd_fast(b % a, a)
        #we're returning the gcd, x equals y - floor(b/a) * x
        # y is thus x.
        return (gcd, y - ( b // a ) * x, x)
```

LCM utilizing the extended **Euclidean** algorithm

```
# A fast LCM calculator utilizing the extended Euclidean algorithm.
def fast_lcm(a,b):
    lcm=0;
    gcd=0;
    # if a or b are 0 there are now lcm for either of them thus it is zero.
    if a==0 or b==0:
```

```

        return 0
#otherwise if one is 1 then it's the other value.
    elif a==1:
        return b
    elif b==1:
        return a

gcd=gcd_fast(a,b)[0]
#simplified version of the formula (a*b)/(gcd(a,b)).
lcm=(a/gcd)*b

return lcm

```

Modular Multiplicative Inverse

Modular inverse algorithm we utilize the generalized extended Euclidean algorithm to calculate the gcd and the bezout coefficients to calculate the modular multiplicative inverse. This one also works with negative values of a. It has been modified by me to be generalized to work with all values a and mod whereas the original only worked with positive values of a and mod. I don't know where negative moduli would be seen but this works with them.

```

# Calculates the modular multiplicative inverse of a and the modulus value
# such that a * x = 1 % mod
# Also mod is the modulus.
# % is the modulus operator in python.
#
def mod_inv(a,mod):
    gcd=0;
    x=0;
    y=0;

```

```

x=0;
# if a is less than 0 do this.
if a < 0:
    # if the modulus is less than zero
    # convert it to a positive value.
    #otherwise set the temporary variable x to the modulus.
    if mod < 0:
        x=-mod;
    else:
        x=mod;
    # while a is less than zero keep adding the abs value of the modulus to
it.

    while a < 0:
        a+=x

    #use the extended euclidean algorithm to calculate the gcd and also
bezout's coeffecients x and y.
    gcd, x, y = gcd_fast(a,mod)
    #I'm just viewing them to make sure that it is indeed working.
    print(gcd,x,y)
    #if the gcd is not 1 or -1 tell them that it's impossible to invert.
    if gcd not in (-1,1):
        raise ValueError('Inputs are invalid. No modular multiplicative inverse
exists between {} and {} gcd:{}'.format(a,mod,gcd))
    #otherwise do the inversion.
    else:
        #if m is negative do the following.
        if gcd == -1:
            #if x is less than zero convert x to positive and add it to the
modulus.

            if x < 0:
                return mod - x
            #otherwise just add x to the modulus.

```

```

        else:
            return x + mod
#otherwise is a and m are both positive return x (mod m)
else:
    return x % mod

```

RSA Bytes to Number Encoder/Decoder

Once again here are the python functions to do this. This is the decoding function. It takes the integer and the output string length(that it should be).

```

# this decodes a string of bytes(ASCII text only really otherwise you need to
convert it
# to a byte stream. Via the following formula. X=str[i]+pow(256,i)
# Thus X=str[0]+pow(256,0)+str[1]+pow(256,1)...str[n]+pow(256,n)
def rsa_ascii_encode(string_to_encode,string_length):
    tmp_str='';
    output_str='';
    x=0;
#byte order is reversed so have to reverse the array.
    string_to_encode=string_to_encode[::-1]
    tmp=0;
    os=[]
    i=0
    while i<string_length:
        tmp=ord(string_to_encode[i:i+1])
        x+=(tmp*pow(256,i))
        i+=1

    return x

```

#This converts the number to a string out of it.

```
def rsa_ascii_decode(x,x_len):
    X = []
    i=0;
    string=''
    if x>=pow(256,x_len):
        raise ValueError('Number is too large to fit in output string.')

    while x>0:
        X.append(int(x % 256))
        x //=256
    for i in range(x_len-len(X)):
        X.append(0)
    X=X[::-1]
    for i in range(len(X)):
        string+=chr(X[i])

    return string
```

Common Modulus Attack

#This does the hard work of actually getting you the plain-text back via the
common modulus attack. All you need to supply is both exponents, both cipher-
texts. Then the common Modulus. It also utilizes other previously defined
functions.

```
def common_modulus_attack(c1,c2,e1,e2,N):
    a=0;
    b=0;
    mx=0;
    my=0;
```

```
i=0;
```

```
if gcd_fast(e1,e2)[0] != 1:
```

```
    raise ValueError('e1 and e2 are invalid.')
```

```
a=mod_inv(e1,e2)
```

```
b=(gcd_fast(e1,e2)[0] - e1 * a ) / e2
```

```
i=mod_inv(c2,N)
```

#In python if you add a 3 argument for pow, then it will return the value modulus that third argument. So, in reality it's powmod(a,b,c) instead of pow(a,b). You're calculating pow(a,b) mod c.

```
mx=pow(c1,a,N)
```

```
my=pow(i,-b,N)
```

```
return (mx*my) % N
```

Small Exponent Root Attack

```
#This works for any value N, C and e.
```

```
#It'll return the plain-text P.
```

```
from sympy import *
```

```
def root_attack(C,e,N):
```

```
    P=0;exact_value=false;
```

```
    i=1
```

```
    while exact_value:
```

```
        P,exact_value = integer_nthroot(C+(i*N),e)
```

```
        i+=1
```

```
    return P
```

Fermat's Factors

```
#this is the naive method that can take O(N) time to factor the value n.
#It's actually worse than trivial division method of trying all possible values
as described in the attack section above but for close primes it's super-fast.
#I import everything that I could possibly use.
from sympy import *
from sympy import power
from sympy.ntheory.primetest import is_square
def fermats_factors(n):
    tmp = integer_nthroot(n,2)
    a=tmp[0]
    b = power.Pow(a,2) - n
    k=0;
    bool=tmp[1]

    while not is_square(b):
        a+=1
        b = power.Pow(a,2) - n

    k = integer_nthroot(b,2)[0]
    p = a + k
    q = a - k

    return (p,q)

#requires integer_nthroot from sympy to work.
#this will solve the hastad broadcast attack for the
#public key exponent value of 3. You just supply it the date.
#it of course relies on other functions given previously.
```



```
def crt_3_solver(c1,c2,c3,n1,n2,n3,e):
```

```
    N=(n1*n2*n3)
```

```
    N1=n2*n3
```

```
    N2=n1*n3
```

```
    N3=n1*n2
```

```
    d1=mod_inv(N1,n1)
```

```
    d2=mod_inv(N2,n2)
```

```
    d3=mod_inv(N3,n3)
```

```
    x1=(c1*N1*d1)
```

```
    x2=(c2*N2*d2)
```

```
    x3=(c3*N3*d3)
```

```
    x=(x1+x2+x3) % N
```

```
    m=integer_nthroot(x,3)[0]
```

```
    return m
```

Appendix C : Real World Examples

Radford Factoring/Decryption Challenge

RUSecure 2019 Preliminary Round: RSA #1

Case Intro: Suppose you intercept the ciphertext integer

[1665116749092532783614517176972314475197848]

that was encrypted with an ASCII alphabet assignment using the RSA cryptosystem with public encryption exponent of $e = 739479573983$ and modulus $m =$

1967790697008364140098628521915198722929959

We first have to get our variables in place so that we can decrypt the cipher-text message C.

First we have to factor n (here they say m no idea why) to get p and q .

7. $p = 0x2d67ffe1b6cc0b5fc1$
8. $q = 0x7f5b77ef8d04520ee7$
9. $n = 0x1696d12ec3f80ffc53651c5a208b6d51f527$
10. $\lambda(n) = \text{lcm}(0x2d67ffe1b6cc0b5fc0, 0x7f5b77ef8d04520ee6)$
 A. $\lambda(n) = 0xb4b689761fc07fe295c2c7127a3ce7a4340$
11. $e = 0xac2c6ad5df$
12. $d \equiv [0xac2c6ad5df]^{-1} \bmod 0xb4b689761fc07fe295c2c7127a3ce7a4340$
 A. $d = 0xa275976b67ca8e241108604e737dd2402df$
13. $M = (0x131d569ccc3cdc5cb86c45e44c5973137598^{**} 0xa275976b67ca8e241108604e737dd2402df) \bmod 0x1696d12ec3f80ffc53651c5a208b6d51f527$
14. $M = 65698332751011215832698367658069$

Radford is using Naive ASCII assignment as given below. So you simply take the string of digits separate it into 2 if the first digit of the set is not 1 or take a group of 3 if it is. Decode the numbers into ASCII text and you have the flag.

Bytes = 65 69 83 32 75 101 121 58 32 69 83 67 65 80 69

Message = "AES Key: ESCAPE"

The flag is now solved.

Radford Common Modulus Attack

This time it's common modulus as the modulus is the same whereas the public key exponent e is different. Plus the size of m makes it pretty much impossible to factor it by hand. Also once again someone at Radford doesn't know crypto as the modulus is always n and not m . For the math behind this attack look into the proof I'm not repeating myself here. I'll just show steps. Also the modulus is ~760bits in length using the simple relationship formula. \log_2/\log_{10} . We know that each decimal digit contains ~3.21981 bits of information. Hex of course contains 4 bits per digit and binary is 1 bit per digit. So by taking the number of digits of the modulus then multiplying it by 3.21981 you can get a rough estimate for how many bits it contains. Also always make sure to floor the value to get a better approximation. It's going to over-estimate the value some but that's fine for our purposes. For example $2^{32}-1$ shows 33 bits even though it's 32. For this lab the `**` operator will represent raised to the power due to how large the numbers are.

RUSecure 2019 Preliminary Round: RSA #2

Suppose you intercept the ciphertext integer

[3348898614019888901908403254933640035246212844961838831090198209986980794381721549119817432681384755697410365192319104804164099565189541033745677177908430898159000425144231080271489963068735078029094600990872466478190741629812176] that encrypted a plaintext message with an ASCII alphabet assignment using the RSA cryptosystem with public encryption exponent of

$e = 927497329847987298271115$ and modulus $m =$

4035789025935566763434217693291904203514985559759202218772232737779637242777118595044390460183072421339720558176591333566629680159420540355202801063004396853930869779589477542063791290354739283500845851153515283182096350655220153

the ciphertext integer

[239640052909589767377717332389707447467040807634556900631678847178249840124011908871933438491107613018104449557989400002181849114477870950132116542696807901617211160049032412890334084433427701567750018959947232564370889351855337] that represented an encryption of the same plaintext using the RSA cryptosystem with public encryption exponent of

$e = 123132131231124141411111$ and modulus $m =$

4035789025935566763434217693291904203514985559759202218772232737779637242777118595044390460183072421339720558176591333566629680159420540355202801063004396853930869779589477542063791290354739283500845851153515283182096350655220153

Decipher the message.

Beginning The Attack

Once again I am using hex encodings to save pages of paper. And like before if you want to see how it works go to the proofs appendix.

$$e_1 = 0xc467bb22cd484f4e7f8b$$

$$e_2 = 0x1a130196ecb7605c8b27$$

$$n = 0xaa5c91f5ba54c6100de462097d18a81cced30e697bcbdafdaa7d084472ad6de5ccc642ab20b4b134473bee8651a1644b5a27a73ed99e8a6954796ff8cd00ed3063ec04c8a2011a2f0c30155f19ca74a37d6c399cfbe4230b0e71bd201f09b9$$

$$C_1 = 0x8d5db860df5564f27e9600b4b21eaae6afa3b8f240b70c75d4c54fed1a4d423dff6ef7f1bbdb607990bc49d5e09df393273250a8411adc2c07533b50054fe0a6c764933c60fa7df2f7dd79a0c42c2092a0deff93064313c657310cc1bdd0$$

$$C_2 = 0xa1da8c8775296a4cf19d60c7b0731b60099ef44e72ad964b764233eb5205ab870ea1f3a04fd221f9e35366900e2c505be26a725996ca34628a519dd635bf9ee185542897405c13488dd06e8eeabad1e0d5328761c12e2d913c64f98e37ce9$$

$$1. \quad a \equiv [0xc467bb22cd484f4e7f8b]^{-1} \bmod 0x1a130196ecb7605c8b27$$

$$A. \quad a = 0xba6ae669209243df6fb$$

$$2. \quad b = (\gcd(0xc467bb22cd484f4e7f8b, 0x1a130196ecb7605c8b27) - e_1 \cdot a) \div e_2$$

$$A. \quad b = -0x57c3282306d561db9378$$

$$3. \quad i \equiv$$

$$(0xa1da8c8775296a4cf19d60c7b0731b60099ef44e72ad964b764233eb5205ab870ea1f3a04fd221f9e35366900e2c505be26a725996ca34628a519dd635bf9ee185542897405c13488dd06e8eeabad1e0d5328761c12e2d913c64f98e37ce9 \cdot -1) \bmod 0xaa5c91f5ba54c6100de462097d18a81cced30e697bcbdafdaa7d084472ad6de5ccc642ab20b4b134473bee8651a1644b5a27a73ed99e8a6954796ff8cd00ed3063ec04c8a2011a2f0c30155f19ca74a37d6c399cfbe4230b0e71bd201f09b9$$

$$A. \quad i = 0x1ed00ccabf5acda10b873005d1ca8edf212c624af9524a962c64efa6ab7630e910e7a6447005ab170b3033d0d7333b5bcacc1ee82a27d0e799585adf5e9b19ae82b254dc9fff6e1917bf58d60317aa212eb1e90c9c116630cd1d84e91443fd$$

$$4. \quad m_x = (0x8d5db860df5564f27e9600b4b21eaae6afa3b8f240b70c75d4c54fed1a4d423dff6ef7f1bbdb607990bc49d5e09df393273250a8411adc2c07533b50054fe0a6c764933c60fa7df2f7dd79a0c42c2092a0deff93064313c657310cc1bdd0 \cdot 0xba6ae669209243df6fb) \bmod$$

```
0xaa5c91f5ba54c6100de462097d18a81cced30e697bcbdafdaa7d084472ad6de5ccc642a
b20b4b134473bee8651a1644b5a27a73ed99e8a6954796ff8cd00ed3063ec04c8a2011a2f
0c30155f19ca74a37d6c399cfbe4230b0e71bd201f09b9
```

A. $m_x = 0x6eb36a2b83899817a1c91dbb22f50349e0e1d0e11b348378a93285ca0d122e4a91a7fb2b8b566bc98ebdd19cbbd4e33ed74b8311590af2fad0be8257d2de33e4f886f0578ba06258bec39ed495d860375aa69b3526f56e62d060dd7cea4afe$

5. $m_y = (0x1ed00ccabf5acda10b873005d1ca8edf212c624af9524a962c64efa6ab7630e910e7a6447005ab170b3033d0d7333b5bcacc1ee82a27d0e799585adf5e9b19ae82b254dc9fff6e1917bf58d60317aa212eb1e90c9c116630cd1d84e91443fd ** -(0x57c3282306d561db9378) \bmod$

```
0xaa5c91f5ba54c6100de462097d18a81cced30e697bcbdafdaa7d084472ad6de5ccc642a
b20b4b134473bee8651a1644b5a27a73ed99e8a6954796ff8cd00ed3063ec04c8a2011a2f
0c30155f19ca74a37d6c399cfbe4230b0e71bd201f09b9
```

A. $m_y = 0x627e01193b81cb8b2a67b5d050180794b6d858faff437fae19f524315cdafd661ae003892832654654cafcdd01d030b296fae0c4d389fc185f9b22c15bca3bc58dcacaf446e9d3a2530fe10ae2e5a91dbe7843dbea0bd8b80cd27483d18ace$

6. $M = (0x6eb36a2b83899817a1c91dbb22f50349e0e1d0e11b348378a93285ca0d122e4a91a7fb2b8b566bc98ebdd19cbbd4e33ed74b8311590af2fad0be8257d2de33e4f886f0578ba06258bec39ed495d860375aa69b3526f56e62d060dd7cea4afe * 0x627e01193b81cb8b2a67b5d050180794b6d858faff437fae19f524315cdafd661ae003892832654654cafcdd01d030b296fae0c4d389fc185f9b22c15bca3bc58dcacaf446e9d3a2530fe10ae2e5a91dbe7843dbea0bd8b80cd27483d18ace) \bmod$

```
0xaa5c91f5ba54c6100de462097d18a81cced30e697bcbdafdaa7d084472ad6de5ccc642a
b20b4b134473bee8651a1644b5a27a73ed99e8a6954796ff8cd00ed3063ec04c8a2011a2f
0c30155f19ca74a37d6c399cfbe4230b0e71bd201f09b9
```

A. $M = 69118101110321051023212111117114321121141051091011153297114101321089711410310132116104101328283653299971103298101329811411110710111032119104101110321091051151169710710111532971141013210997100101$

7. Now with M you do as you did before divide it into chunks based upon the first digit of the chunk then convert it to ascii.
8. M as bytes: 69 118 101 110 32 105 102 32 121 111 117 114 32 112 114 105 109 101 115 32 97 114 101 32 108 97 114 103 101 32 116 104 101 32 82 83 65 32 99 97 110 32 98 101 32 98 114 111 107 101 110 32 119 104 101 110 32 109 105 115 116 97 107 101 115 32 97 114 101 32 109 97 100 101

A. M decoded = Even if your primes are large the RSA can be broken when mistakes are made

9. Thus the answer is “Even if your primes are large the RSA can be broken when mistakes are made”

You have now captured the flag.

Fermat's Near Prime Attack

You were given the following key it is your job to get the decryption key out of it. Then decrypt the message. The cipher-text message C was given to you already.

```
C=0x9a15ca9b78e1f0bde8bda1e98a1ece19a95ace7354f8df44532ba4b4693694dc56d65174e5f
7c4e53c061ccc0e716170199463fd898474120c1873413c0700b79f2e935413ee6e5678029fb385
834dc601ecf864c9c79f48f462dd4af7c473f6a24f505aa80a8926d62330bd41089e471def17662
591ac5aabddf9ce9d51228fb9223b6e8dfe2d7b16b5c44c9722c9b7d2e84d442e9af7c966e1d089
34ee7d815906f5085d39443af4d81537cfc0d4173ccaeba89c554b421ae602e1de9d1ebab6fe0bb
ca268198e5db65c4b14accaae72fbdeb0a0ec13351912e87611604d71a5ba85ab77bd62135232b70
eafe78041bf32bbb05bd9b2b64b3132b51d75a333
```

Public Key

-----BEGIN PUBLIC KEY-----

```
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAx9EL9EgTc9vE81nYE8+O
8YERRhBZgkfkYAdsRloylStvD+cwZXAhxljd/XWZtch00mVkzvzVaCAk87ZrtnM1
AlTFQ9ycjT5QeQ/pblT8WeT5ygi1q1V7sD7Ad7rSS+3W0Ja9qYkC7qvsEbFh0npR
nDouPpb6JBZk9HE/X7Lpn2CsG73sLU7ssViS2sXtwpR/kyne5ccYeHYtmtdTdLo+
H4gJfJHaRKETphkqS6tn3rJC2N5czz9AsCP1CrDcGQQEcI4mbthmFzTY81bFybLK
HXpY0VoJcQigp67tipQ7RR9KcWvuboXAHPhs56oVqM9wjFkUo65oiY1RWdh5h8Qd
dwIDAQAB
```

-----END PUBLIC KEY-----

After decoding we get the following vectors.

Decoded Vectors

```
n=0xc7d10bf4481373dbc4f359d813cf8ef181114610598247e460076c465a32952b6f0fe730657
021c658ddfd7599b5c84e3a6564cefcd5682024f3b66bb673350254c543dc9c8d3e50790fe96e54
fc59e4f9ca08b5ab557bb03ec077bad24bedd6d096bda98902eeabec11b161d27a519c3a2e3e96f
a241664f4713f5fb2e99f60ac1bbdec2d4eecb15892dac5edc2947f9329dee5c71878762d9ad753
74ba3e1f88097c91da44a113a6192a4bab67deb242d8de5ccf3f40b023f50ab0dc190404708e266
ed8661734d8f356c5c9b2ca1d7a58395a230aa8a0a7aeed8a943b451f4a716bee6e85c01cf1ece7
aa15a8cf708c5914a3ae68898d5159d87987c41d77
```

e=0x1001

For this attack to work we need to factor n into the two prime factors that make it up. To do this we are going to use fermat's method. We will need to calculate a , and b . So that we then have the prime factors.

$$1. \quad a = \text{floor}(\sqrt{n})$$

A. $a =$

```
0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47
d668431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9f
d98518f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1
a4191c52d22386c3b98572c779e8029ae0f61ed8c2f40603
```

$$2. \quad b = (a^2) - n$$

A. $b = -$

```
0x1c4573dc24a9386d7ca1157035444304d92f37de8bf0ce8299498816bdd65f617b48
facd0863c033dc35fc95dace9a3f2e34389ce1bc1304544f3be9b89e08246b50ee4d53
fb30a31e4b775c95db022470db437956f61c719f6c0e5c3edaf82fbef75906a5d8767c
3483238a5a4470d87730ae58ef3d00535c1ec3db185e7f96e
```

$$3. \quad \sqrt{b} = \text{floor}(\sqrt{b})$$

A. since the values are so large. I simply ran `is_square(b)` and checked if it said false or true.

I. `is_square(-`

```
0x1c4573dc24a9386d7ca1157035444304d92f37de8bf0ce8299498816bdd65f617
b48facd0863c033dc35fc95dace9a3f2e34389ce1bc1304544f3be9b89e08246b50
ee4d53fb30a31e4b775c95db022470db437956f61c719f6c0e5c3edaf82fbef7590
6a5d8767c3483238a5a4470d87730ae58ef3d00535c1ec3db185e7f96e)
```

II. False

B. Thus it is false.

$$4. \quad a = a + 1$$

A. $a = 0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47d668431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9fd98518f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1a4191c52d22386c3b98572c779e8029ae0f61ed8c2f40604$

$$5. \quad b = (a^2) - n$$

A. $b = 0x1299$

6. Checking b again. We get a value of true.

7. a

```
=0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47d6
68431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9fd9851
8f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1a4191c52
d22386c3b98572c779e8029ae0f61ed8c2f40604 and b = 0x1299
```

$$8. \quad p = a + b$$

A. $p =$

```
0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47
d668431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9f
```

d98518f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1a4191c52d22386c3b98572c779e8029ae0f61ed8c2f40604 + 0x1299

B. $p =$

0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47d668431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9fd98518f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1a4191c52d22386c3b98572c779e8029ae0f61ed8c2f4189d

9. $q = a - b$

A. $q =$

0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47d668431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9fd98518f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1a4191c52d22386c3b98572c779e8029ae0f61ed8c2f40604 - 0x1299

B. $q =$

0xe22b9ee12549c36be508ab81aa221826c979bef45f867414ca4c40b5eeb2fb0bda47d668431e019ee1afe4aed674d1f971a1c4e70de09822a279df4dc4f041235a87726a9fd98518f25bbae4aed8112386da1bcab7b0e38cfb6072e1f6d7c17df7bac8352ec3b3e1a4191c52d22386c3b98572c779e8029ae0f61ed8c2f3f36b

10. with p and q solved. We can calculate d and decrypt the message.

11. $\lambda(n) =$

0x429b03fc18067bf3ec511df2b1452fa5d5b06cb01dd617f6caad2417736631b925054d1021d00b421d9f547c8891ed6f68cc76efa99c780ab6f7793cd111ab7197169eded9bf70285aa324c6fec8a1a898ad91e3c7293abf957d3e46194f479adce9e32daba4e3f95b3b209b7e1b341364bf87a8b6b221a6d06a753ba3351f0bcf1622880d4a55ec25643fc1be114c246ba106c2ff8d6101789c293dfa1b500a4e0ba599eede32b51ddd301b3187997ec5ee4032944ab73359d9d342c7ffd2577ac996c8555645afd6ce665a1b81da74fa98ba6c17b379936460ab92f4150d23d76e5de67cfadccb2d7cca445e3c208cc11881eced7fb5ef2b1f077e336831

12. $d = [e]^{-1}(\text{mod } \lambda(n))$

A. $d = (0x1299^{*-1}) \text{ mod}$

0x43ba81f8cd2d7ac9f55a1f0bf4bb17d298530892a62458c5fa1ec9fe96b82a83cc6371939fee8024411f0db87cdf7703bd1b7d2e03d93a301b1658b0ede8e300e546f017a920df0c5bdc9edcfd05845aeea82287a1899a214543ebd17152b74d66ea916313c3e0ae79980dae00afe155b90acfc0510d4661f5d12510050ed72c4e16f6f23b9e1589c699f3d5543dd92c7191fd1207561415428fde1a5fdafeb75b9120708151625e8b3916b7d4d96884519bc64b905f9ca4fcateda2147a7ecf96ec2d8a288e184cf6080ddcc1a7acc82b8ef3e27e9b31c67bd652e06cb75698b1504939dca0b9c5ba98429698efefd058f97d4fe367ab041ae3a5e141701ca1

B. $d = 0x132e8a46cd987647b0e130b866dffe8312f5edd3356a33f9e2fc210fb21f93d69dd7bb64a69972a9e1819c7f6383d305c9024465a4c2f72ade664519c3700dd3a2f66a1420b7c41f756de88c677c621a3aaaca3eafdec3873fb2621d7d0ba2c59b253ea05a025e665$

13. Now we can decrypt the message C.

14. $M = C^d \pmod{n}$

A. $M =$

```
(0x9a15ca9b78e1f0bde8bda1e98a1ece19a95ace7354f8df44532ba4b4693694dc56d
65174e5f7c4e53c061ccc0e716170199463fd898474120c1873413c0700b79f2e93541
3ee6e5678029fb385834dc601ecf864c9c79f48f462dd4af7c473f6a24f505aa80a892
6d62330bd41089e471def17662591ac5aabddf9ce9d51228fb9223b6e8dfe2d7b16b5c
44c9722c9b7d2e84d442e9af7c966e1d08934ee7d815906f5085d39443af4d81537cfc
0d4173ccaeba89c554b421ae602e1de9d1ebab6fe0bbca268198e5db65c4b14accaae72
fbdeb0a0ec13351912e87611604d71a5ba85ab77bd62135232b70eafe78041bfc32bbb
05bd9b2b64b3132b51d75a333 **
```

```
0x132e8a46cd987647b0e130b866dffe8312f5edd3356a33f9e2fc210fb21f93d69dd7
bb64a69972a9e1819c7f6383d305c9024465a4c2f72ade664519c3700dd3a2f66a1420
b7c41f756de88c677c621a3aaaca3eafdec3873fb2621d7d0ba2c59b253ea05a025e66
5) mod
```

```
0xc7d10bf4481373dbc4f359d813cf8ef181114610598247e460076c465a32952b6f0f
e730657021c658ddfd7599b5c84e3a6564cefcd5682024f3b66bb673350254c543dc9c
8d3e50790fe96e54fc59e4f9ca08b5ab557bb03ec077bad24bedd6d096bda98902eeab
ec11b161d27a519c3a2e3e96fa241664f4713f5fb2e99f60ac1bbdec2d4eeeb15892da
c5edc2947f9329dee5c71878762d9ad75374ba3e1f88097c91da44a113a6192a4bab67
deb242d8de5ccf3f40b023f50ab0dc190404708e266ed8661734d8f356c5c9b2ca1d7a
58395a230aa8a0a7aeed8a943b451f4a716bee6e85c01cf1ece7aa15a8cf708c5914a3
ae68898d5159d87987c41d77
```

B. $M=0x41682041682041682c20796f75206469646e2774207361792027746865206d6167696320776f7264272e$

15. Thus $M =$

```
0x4920646f6e2774206861766520746865206f726967696e616c2066696c6520746f20776
f726b20776974682066726f6d2042536964657353575641203230313920736f2074686973
20697320746865206d6573736167652049276d20656e6372797074696e672e
```

A. M as integer is

```
3195592863083917568262692656039222883937434538376160587203344639800560
3998223038265467595809324586726505576805690243814487218483680893449058
7648675518132973302838082889275625659741866415640091437770888610045419
76616436704523481167295903867828332334
```

16. Now after converting it back into plain-text through the function `rsa_ascii_decode`

A. I don't have the original file to work with from BSidesSWVA 2019 so this is the message I'm encrypting.

17. I don't have the original file you had to decrypt so the flag is "complete." In reality you'd have some magic phrase or something that you'd need to work with.

The flag is completed.

Cubed Root Attack

Given the following test vectors calculate the original plain-text.

$n=0x2176899ee2dfc6b5ef46a65d1a130a9a0156008d4db4099cc1e6284f58c2619$
 $C=0x8d9ab41d50ccda8bfa0adb670769290b1ec3322a58e196b61a83f900729$
 $e=3$.

Solve for the original plain-text message. Seeing how e is 3 that means that you can try an e^{th} root attack.

1. $M = \sqrt[e]{C}$

A. Plugging in the values we get.

$$M = \sqrt[3]{0x8d9ab41d50ccda8bfa0adb670769290b1ec3322a58e196b61a83f900729}$$

B. $M=0x536563726574204b6579$,

2. $M= 393826705131131749754233$

3. Now we have to decode it using the standard RSA method

A. (I'm not using Radfords method as the numbers have to be much larger.

4. $M = \text{Secret Key}$

Thus the attack is done.

Hastad Broadcast Attack via CRT

Throughout this section we're going to assume that you've already done the math and have a script to work with these numbers. Thus you're solely going to get the values as they come out and are shown to you. All numbers are hex-encoded to keep the sizes down. This also assumes that you've read the Glossary Appendix.

The Given Vectors

$e=5$.

$c_1=0xd30c93811204c45b98da8d98cf9829551cfa4464b378f7dc700e2000db6173b50c22895e3e82a6801f6328eceedfceaaf7981c8037b2480641200dc95f89802849bc95a04093f0b5c7ddc95c828eeb66b9c2d614025eb2d9f13d4de039b2a7a7459b2c14c9dbed1324822e3eb294dee8964$

$n_1=0x2cec070086977dcb0dceb4ab27dc35fba7604b186f3a010e34bf9aeafd0798c4b2bad15261afd19a01d908c4b040c0ca2888a4189380624f08cf7f2ea080e05d8f11a559e1bb04f343619abf15c953db58b86ae65e9a356805ab629b643b06d462fac63013400bd5b6c6810e1083b7e27699$

$c_2=0x1d6906e44743efe3a2c7e86d58cef0f2ca2a6f2d61f75e8d2f295bfd186dca84db3ec61f07b55c24e21826ffd099917f984acab889226f42330257830e2c7ce92aabf9544110c221c2c7cb41e5c0ed9af875f60c7e2658e7b185a3a394813783c6528cdff71f8ced2db14e0d9cb65b5e986f$

$n_2=0x39b300628764fc7af4ab0acc50f1d1115a3bc0642bcc0836670e7597afc154d688864a29f3123212873ab830dfcbf37a8680483a6dde8ad9161886779b0cf6fd551a647001205c08af07afe20604506411ca662f3519ef5e257d4b9d8d7c2f293704456b0ffaedfe8b8bd4f88c0086df09f9$

$c_3=0x16fed13cee720525d22a79aecad543c0bf7c1c2d6c436441d914bcde12f78171f6ffa46981338317c8527d7e6a0c515a17c72507cb2e4fa5bafc2f419af117278de61ef207f087e610d28ad433fb2e70b2dc9c8a1568fca86d796aea8569428602ab89a29b926d39094deab43fb8c1468b4$

$n_3=0x23d8aa6c92d22590f0272e332f237b2fd5c4b104be32a9738ddb65392a366f8fb78385a01d0f7d5eb4536f8cde373e47cd40472eb37fd9281a2d096e469a8f2ebb0d1694108b8e3915a37dd4e8686c880211d2f9d4f7d88bfb73ade9be9bdb54334417c518abfe0f7a8fe52af6e9a164f5bf$

$c_4=0x19a60b461c4a6618bc6f8f5bdef670a3d6db19b5dd44fc265a8dcc19ab7ffdf2e3bf9fe17ae2e381b71a436599b2325bc8fdcfdaed99d95ee46452250b4e127bff45a5d32a078191fb0c74e2bccd3eca23b3016fb376c8bfd32ad187098025cac4c231292b280a2e334f0aa0aa9348e4154d$

$n_4=0x30e9d69bf662d29e23ea8a5eeb738bc23b5a4bfa4e5b28d826fbfdbaae019e34ffc4986f6e860d9caaac2f72db762ae3e84d17b3e891edb783c7f7dc8320b354f41fe57db758c7b0b0df21391f91a812ce21ef1eacd5911229fd6abec5604d9fcbdc2a55125e9aa9ab124cef8500f2974d19$

$c_5=0xcc54c87dca21a18f63ce618b48c4ba020bbb3b9100abf254e0f0e85b4c1e7ecea505bbb05d57e8ff610cc4f712c0e0f173eb60f0f5c5949570c05b4fc03bfaf3327127a3b53434ad193b4c294caa422d596483daec7c37ec5e2e4b24e0f7174f52edfe0a5865b7db7191d27e9c4bd79ea5d$

$n_5=0x179a8ca0230cf6ccf082383f337f321d34faaa542ff518d7ebe8f56cdea6046b1ce225e64ef3010330889f4dc2980379eaaafb2d8956030c810dab398230aaf7e8ef20e51c8f1f8498f6ec58b958e954cf3f407078a4d76d5255ad7b2c9d300bbdc410716c759e3247dd7a0a7035c915ab04d$

Calculating N_s

Then you have to calculate N , and N_{1-5} because $e=5$.

$N = n_i \cdot n_{i+1} \dots n_k$ where k is the value of the public key exponent or 5 in this case.

$N=0x664d3373857d79c95fd4bbdd868f6fdfd736d77260211ed74f1def9907467d88482d9e2dd91099aa32275fd34b88b9df5e91e07b05b78512d8538056340aa801c714a4ed4ed5989906beda9b1c713a610edc19099d60e86ff23f2bcaee398ec80b4dae9d5a21861a6f52823a8d0bdb49c5a4359719c0ad1e9eb63cba6c4b4df00e06c89cd6f3d12a109d7e09aec343bb7c5a3e39187a17f1686360658161fd3abdfafa0cbb826ca4bd04bd9ccb6f64bb284bfb603ca41321231db9bac0448701f8076f80d1d6c82498a1bdf03494ed692c12f0ebc819f2a4533f85dd9391094acc910d9b05f1f582dd87102ea085016ed3c0907a89af8f0a98c7644a1b7c1df8f26dba60116490bd1195ae8829aa278c15de941a0d7f18b164c3fa6f63fd2d19ddb4f1220bb79477c6d04b50ebdba96a7c784842c55397e51e39f627bd586d8c38dcf71a3cc3878714dc8d4e843fba596b7f71b31a562b01a8597867baf6991767d739d0eb0327fcaca51cb89c149f63dda023d0318499119aa467867f6a2aed9d581469d5c7efb2c99ef6f660af6c9bd3f546ce8674e4bfa5c5c928f96652f5466cfccfbb101f6818811da09b0394609347f4b0d3e89900343a5fc8193289d871d9c6332095fd7faac544766238cd5cb7fd9c21a5177d843ff2e54757233d673e9d82e44b457a173e7b4137d9945eb43bf48ef403bfc247c91e9e665e8b89d1e3851f7e1fb88ca6af66a8fd85f90642ed8e2058f627f7b491db0049cfe7e65b1066f78e0cf91bef11a75cccb07526f76abd91acd88552b$

Further
$$N_i = \frac{n_j \cdot n_{j+1} \dots n_{j+z}}{n_i}$$

Meaning you're multiplying all of the moduli n except for the N component you want. So for example

$$N_1 = n_2 \cdot n_3 \cdot n_4 \cdot n_5$$

$N_1=0x246fe35475031b158989ac87262e43041eb15a6b9f8436f760e8c033c789dac9967423a8c94025a08c7221588b31ae2d609a9270b5e92c948140bd6feba7b2047d457ccf520ff6b90509cf8d4a60cc5e78123088e10fd60a8c5fd6a43688df413bad0c1538f31cd9e11c508e12308538b347f6bf$

2141d620c89f43a3f2fea30e8a72244e8a585ad80b1b2e64f7618e6ba626f02f6eee4943e27a844
8137126bbd852fb3185bd18dfcaa9b96f38b9adfbe3ed67fbe40b82c5ea60e9baeb0520b15c2754
18de3442531849258ec6376543028ca97d8ff2c38d64e6052bec194552ff728b7d43b76ba43e526
d8b975a3af24e207f1b6186bb8e929d71901c3b3b5e99bdbd2f815209ec66909394fa646efb3788
2f65f0522819694f25f37b0931939db4f0d8c96478333d499c302244d21926e3b0c084b0500f2b0
a4ea391818524e05dde4c5d74699dc185aa1d8e6cbdd55fca4aedf0c255ed2cb41caf567d50723f
27b0001c6d04e0c1112088c9b4dc376f02a5013f1e577acbd9dcac33fd0126dcde2a6108d23c975
99d204f0edca75947617039be46063df337b8c8ef8e25b61715e384ca5c848b3a29299b911277e6
8b893d90b6f89774b5ba3fca73732754f52e4c37fb3863

$N_2=0x1c5e4488740ff3c816548d5c41164a6e6f51b55cdafb14ac8499089d4b04661219a923dc20$
fcce6bd6a0bb73c991a811ae58e764be17c383a328b08b9eb6e7592b824e9fa477a0710f6041184
d828e0968e4f39b032f188163a22cd9602eaae665d35fb2984a9c46fd2a7fc01ec817f095e4e261
0e257f43b446277297fd54c05503d3379871fe5f7e317b90f8af17299d1a0b7a6efe79d26704893
2daed81d9e5c6d35ef853690683ece2a56af071a3b268897bb862ddc3101295b70cd392fd7b2a17
12690a2cb107336195f2d9703d0f3176a2431bc89d8d8cca656be8e1f1e23bde0bcf008b1ab145
6e9ad1e764d1ca6cddead151d5453bee5c492a27ed02ae0aa2bff0c7043cc1262046e613d635a4d
02970020420fa07977cc60731affd12402652b0e9faff63e9d799cb0d3942f543022db3bba52e18
e0ce89d62301cb0c0f7faef76bafbe3d6e45c0f6ddab37271bbb5710d017884d9968ccc1866b4cb
af90ebae63d0f4fb1ab79f058e710c762200f4e5c5e57c1493ebb54c40964344cc384915d023055
524742fce6adfa111a0ddc2008feb4297afafe5162fe61b9a7291ad91edef8e9e9a3201ee50ce2b
44a53ec6ba543c70e9032274879aa938f41ce178c4c143

$N_3=0x2da9894c42a741abf96b7b9c0399b0266c6ccafb59f34b46e95953c3fecab3dacd609a9e0f$
ea8ac12b8b6c8a5119ca76fadcb17baaf0a7ed3996f8d1d463cd64b5bd7166457016bcf3de6fcda
4ef9dc143e1a77fd2a9932211e047e8b17dd95f206c2543c1eb39eed9cc7790fc0aa4d8075d9ce7
47348d2dc038671d7de4975f785de526cb09f56ad7d89da1039a6da784b0e8a609f9a5e2b9c71bf
b3cc50d82c1433ff2d64c782431ab3c152fadc3d2e6c88e3657da787a4f997fedbaa765e46bb6cd
a9cb894019242cd8b528da5ed524c2eea031b8838f0c0c63deb01b0f646784678a988f82a39b3a9
fa7c70fa6f38f69c8cd355fc013ba70b12980b0eb9cbe2de6e960fd47043224f11c41c59fdfe814
83a2d96d1476ad9500675add3e1aa6d3db892baf69d18d0725cb08f7cf3fe5f2eca8104dd626b36
44728bb68502094b83d8797ea71292131f1e198866bdd26779f708ce09dbccb397b54394c7a2fbd
5f8b0c1a27e8bc750ebd322d0e6bd002587de75d1a1829c147e04cbbc33575880aa98435b482ec3
5f9d246446a6031eca7e5b3c76e3832c0bef33868382c07572e17ff9459f59ac62c9e2b64b68a10
0f9b397dd9ac406d2b43f72609ad0f573b21b811bff395

$N_4=0x2176b5f544ea7b1f4ccea1495d3a47f4f0b2f4cb10cc0fb5858f5e37e02fc26b23662184e$
fcedcdfb777d6d16ba34ecff352cac020820e49fba700cfe8178133e7fad934077651b3fb0d9e54
418b27baf8439777d997fb2848f9d4753889c6a90d69729cf95171fceccec17a944cda758d17979
bf46af437f95cccad19de85488d2c87e2d17e09ecbd4343f9f107c52ea5af8172a7ccf75a6e34f7
88fc17c7dd239f3a369d3caec1ed67b12f14d6cc87a32cc0900ccc6e44061c06b69c3a84f632e13
8da8e485924ec276112115c094ce9f1b59b846cbbb3f3b3cec46af2204ac4e8c09a4a934548fccf

8fec7dcc22de384b45fcfd4d98573d993d630abd317131fb105e65a7b880644d0c5a6f7feef1b0e63404c82316ce3748703f444f5903fb210484e543ed61f745d4b2ffdaf853c54c90b6895c6d0cb7f8e1cd62602f82db9b7cacaba23ba85cfff0d3a90bb06429328d3fd09dd687d02455677db18cef950e3b000cc3f0c5b7a30f6dfa2578f4886ab05a99782db1d46afe08929e0050fc1658439e15f75736db2782ab1a3be3435020f3faf48f81045092489ea97b405d53dc40c1316679a6fdc0f23eaa0e96e14743a822a0bf1381940b81aecbe156bf8858b3db8e3

$N_5 = 0x455899dcb83d636f5a5731203c134c1b3a4bceb8f326ebf768983a7c423d34c6cbae4f6b2deaea3148928b3243f91870d123ae9eca296bc788b644c7b09a3f0e6dfbe54e6887584f1a591b8ca07173c75ee52a72c1b316fbadec2826704a5aff8a962ad995d5310babb48912b8df3aeb4afed04b43a67ea855a8c166c97508374ff0eb8a7d0fb7b213e204ecf2db18dee30d920549287d5261471e80c9b50acb2733a43c96cd00cbebc95cbef4fea4d445013868dcba85f7aa2eaec3d4acefc47cea6af56601052e89e7d6505ac34dc2351cc9a221a32284b27616ea29ee184fa02b63d6885894e3647c04e9204b69951ba63b1080dbda319e70ebcd421853632f911b86308bf4ee38a7d88b894bf81a67e7182400ef951b001621be16c61f63ee8597b8aa03a17c0efcad5a8ceb5c53e2c87f959e520a57605a4c7b726d326497f770dbbef5c29c69d1b6ac30ea48334358cc7584910bf1b9946cce1cad50fc99f8a0d6de8531fadee7664525257d7a24d6ec875ddf4c29fffb213392220b74a96e150d9fcb7e79a5028b3e1ebc6741db6ba28d0e5a37d9aa0612bfbcb4b9b00628b196eaa5c926a0a57e79c7835fa645c5e43fdd52e6399020bea83c59b3438fb2f351d9757$

Calculating d_s

$$d_i \equiv [d_i]^{-1} \pmod{N_i}$$

Meaning that $d_1 = \text{mod_inv}(d_1, N_1)$

$d_1 = 0x117d17785feaff5a865da9e2b07092991963e2c7e6be6c4a11762a074f7dcb549dc1e3afcb541c59351171266981c72e790ba2d8cd519bd024ed80ba9a4379cd6652b321a598c322d39c6a36d7a681663d887823d884855d591585f323e678f16ffa38bc41d82a5bcdb5a9df63fd4c0a1d84$

$d_2 = 0x1e8a72edc7954289985e846bedc24404c37261e88807e1bc7394352ad67b78341932409383c5268269913614b0d42e22a855ccf8a91c727873c36265286f22454bc3fd47916f4db0335fa2158e503dc8245a8ba87536525c9ed01c2dda8aa78f8bbf71b85f34468ed45b3bf31fe1ca6f952a$

$d_3 = 0x1adeb547447989faf03a5e0561d0b82b3cb5cbd5e166dd843a8e7b623f23ae460764b1b971cdc42b9fb4adfcd13899110517cbdac57cf6ffeba44210dae3aeb18f656dc6718d3bc24d65b9b2e36e7169944c8d5fb1d471310c36ad0da365a6421f7e44c761bf756dec7eeddc49aa66176cc$

$d_4 = 0x15e113f92034442d96ec2526ccadfe4df89a50f36f4e2b2003451668ccc3e45bcf7cb8e5fc3e822703bb7f43744cf548aaffa85b2384668c1135fb08b2f813b71ba0d37afd241e418d85076355537424901c429e633bffca5eeffd7f793c4f525330c6e283a93fe62339d4cd74b789fdd05$

$d_5=0x14e08d4dfaadd1ca7e1e4c3e1a417710ee790c470c58312aeb93cfbe231bc88b5973e9314c773b50a9bd93798ee93820ea394f2da03d8c9e11db5f8ef5fed7b2052d07ce31abd642b176082ce5fbb459eb887abbfa97497236a2dc068b178e66804eb7abeb8957285ac7ce6d6ddb1463f5fb$

Calculating the x_s

The components x are calculated as follows. $x_i = c_i \cdot d_i \cdot N_i$

$x_1=0x20d57d0a65d5a15d720c223aa086fe54a7f70b05f94efe41c3d472e66737d1dea897178de35cf3c78c3d2a28a91e0a09761b0382a5a7e5f7d198321a0ba58f025b4fa392e331d3991e59206c7af28c1868ee7b16e59aaae137b0022adf48a51ac6ccc73c7131b525be46d9c96864685c37112cdf32c9417863d18b868a618b6fca43c58432401c30c0d25b442dc2ff4d73cd8d816bef072ee2ed17da2aafab2666444f6ce473fe851a9daf51bd200f6df2c19d732652142774c1d889f925f66511379fb674b76e7a54f27af0d7f552e6a2fac4b6ee71ff0188c6cc8eb88c7abe80630599dae0b2cf6f4bba24ededd5b4b9b6464e6549d86b4fccd9996c99ecf0afd74fbe4d335b4c1ce65e4c8701710da79b92a29b547aa4ef53273c7c65da063d9fd8f5f912b7852d6d26f400f491b2eece6c4138cbd5af97b4e2c69dc70e06d9816e2793f1810d90ee49467267fa4ae664ee4607c84966050cc15f352d4ac3659fd0000108f7a974c3846943fd361f92092ad99bd3b6a1a720716ae256d58aea3efc5884c31d08e81ac30341ea238dbb0e434f72a0b5280a27c29d281a64f7636c539619fa2a9e62fce9cde57ec43df26e39daa8116495d2c54d1281683c7f056fea4d528ad55e6428fbf9ca880944b604754a787bdac6ca8fff9d91a6dbc6b55c3525d5a97f943b48660cd231f5b553e4eafdd7c97ae7a06d4e51867f8896e7cb7cf8b73ef9b936ddcc902ac374b04bbadfbe2c9bbf8e6ee715ff123863346ffc9ada46bf420ead18d83f9e78e4c4438c6aca8cfe7d5386e9c965259bb23c18beb0b72176e2a5d2293f13c24a23dd9a4c3c0427f351023ddd0ce6474ffbb133d9522ae47876a026f27dd0b04dcce75f21f5b90c0821b33d0f3f55fb688e0e9fe5deb50e84da27e9cc4aa75db040aca793afea89afa788a34b523c916e3655c78fbc524d646bcd58b0$

$x_2=0x6388ba13b8bf3f29d36c4b96e981bd80d5d0561c3efe6455bd0192edd6aeef698e144291389da14e410fb564b1b75d46f9e63977cf55fdb9c6473e48341ba208065fbf2d3684a008725c7b012fec6675398ad8fdb0baa3868cd8b3da7a56a8620b561d0ee134a403df345e83771529ba683bc04e42d44a09755082679909f4991d8fb1b67d0352f2e49db94332bf8bc4aa6a6a8c891f7d8c8e45df44795d13404727cf1dc38c76f00ac72ad9509e9027e8d586e02c88e3fcd2be83493d9ec45add769f6056b5f4a6e4008db4ca4b125e350b306111bab7280b0de2d2da824f09d53aac1947cb148c7341bffd11e5fad0f0f8c5ad6edab458f4f076f7e99814247d9a90619bb3dc3d62dd035f6992bf2a83459d193c82f09ba847cbe530ff48a07c4b8186f6c997217cf6d6e5b2c0c768a5428aa726836fec27286360a078cf98215c9473ce0a02cf49bbaf7f4ce7196d1423dea0849389cb6d97f863f3f3a76859295a38ecabdb5b03edd21863a016a14a27ce082ce9bfba8d7d86853921886564f72b0c9aa0236f0bb1f23b247e84fbfc6b6ef28a55a85cd0234cb9d2d84dc8c36a0886fd7567ee2dbe59e12b3ac0ceb45edc7055867cceaac1bbee37db2038c738492423d79cda6125c80d3ecedda79a7bb1f180210bcced200099b1cc2689141b4d9f121cb2d3750f0b9a7561c6187bae711c37029ffba950091fd42aea d889782628ad3fc1cc0c9511782e31e066b42b0d8cf19bfcd735931629afca0b6a2590fd22151b3$

0a1a116ecfb3e4c04d83c94c864dfc50e1e1a3d50781e39d1f55e8ab80169e6b237a2c492f2c539
4f3b2f7ae0e8f976735c9d66ed8507a1a577c7f99d4140e049b703e2d335fcfd0a9419da26acbca
d85c69844ede91d06a798045ed029be5bb0a2b5dfb14666d6f9a9ffc617a6cd9e5d82ef69147ec0
ee3d5ca99b871eece2c8a0db22

$x_3=0x6e3602cd57c1b5ba8847b1ce80d38d62e0c799a5b6e8e4d0fa32e516bb701926ed284a84ed$
 $e2f60ddf0a0eb8b593f348886b935a7e6dfed4af8a1950f11864430c8bdba092466338d99088fce$
 $a7679cb17211d1639ce113d296e73f4d7158ba17ad64651295416cfa982fd4f87376210537a7a27$
 $d945b3f34bf6adb49cf62021372242a053b5180e35e3f2fbcd1fa071058942c2adf6469d25e9826$
 $3b96a2ad7533c4a8694ab52806ac09d81df3e6164807ee9e39a887675652c6fb0ad75e9401d9070$
 $9bc5f58644acf163629d233192c57c8c0369d7d1f0c20456e67d01c21f57a7ed31c1ca497a5b368$
 $119e0477c4e1be1d4b6a43de1c6cf1a047e539d1862d335128155dabf0d5d67ec252e153694b156$
 $1838210eca20fe7237ddec4b56db78492c56276cbadbb81157b1d00e14ab5faa6a4d5a87a310293$
 $17aa4a9f3b16d136f3ab26ba63d2ea6b79fbe79668c244df0e7a50a5fa519ff9d99e1c60f28183d$
 $7bb72e26eface7a787198060a48a1edd735a9320f418fa124d688847f14194340ad314f607e4180$
 $6ae2b92e621ab68b9db2d01a3953de1ad4bba064879f8c7a20c7a4d4f91c8e928c3d7c875b3c17f$
 $c4df232b5877da003af589517ef5854aee066dace05f14ff9a20d51a1624e705ca7a7e331f60019$
 $ba0a97f47704f8faea58f0ed2b4da8439b5635bc21d34e748d6291cc95b6f935ac40c3741098ea0$
 $2fc137a0df7ef3449ddea697facacc059e5a3d7ba71df23eb66a78a8a1625ec8eb6d0768f6a5065$
 $e26791dd80876fcdc6ae49071a67dcb316303a23bf09abd5082f96f1fc4c9870d4bef4a3bfc94eb$
 $e6ae27fd5d83b279c7f7ccfd64aa43b3cf338dec16aec78f8e59cbaffc0baa4dd32e49272702130$
 $b7b4d940f3286dcb34ed8c049e366474e39d6b6ee7e46dda880d665fa2d1a024e927a569b3121f3$
 $c429fb70a9b1ff2d6d53d8430$

$x_4=0x495ae090a25aa724b1613d5c4280a9917747cef4a1be3f7261740e4b6dd88b23fe3342b7ae$
 $293f747fc11824008a483fb5dbc90652005ce4161934a304dcd86defac551e55ba53f57a5d7830b$
 $0c79dd2b6f2e97668c0b2aa53898373b6bef015f4fa08bdf6e1cb360158b5c293204c7433fb48aa$
 $5bc0429add23ae6646eba69ffab8405a50d190c0c5378879a2e2bb7efb9e381243ac401d45153f3$
 $47e1bed8cb313896ce881a1d19309201a7d1a4b29a08e0fab516e163419170b3bc2f7942a740932$
 $5338361bf20d18c6ef0573f7734e75fa1047df145439a39e23dc52e5395693dc276edd2fbc4177a$
 $b3c169387f45fba17f8441f3173d0224788923fc599aca6604a6f100edaa8049851bce07b2cc5f9$
 $58c96cf9120d023886d3c84b6f1771c5c312ea4561489ebc51305c43286233c292c67aafb32eba2$
 $d462add40c2f52dd28747e2330bb9d8b0be213e7ae183d9dca325b410fb247838516a39e5fbd3b4$
 $863cdc424ed2065d4bfed890fd3da3516aa29cd55dd9205fba7c6af9a94ac8c274af7db598a87e4$
 $16502706264caf7c275bfcc0f5f089cd264777252bbe39cd6ead953ee20bfdcdfd448d41f62ee60$
 $be75ccc7a5c9961927a8894c8a78ca53e06b4e3724c26358ac891d7c635eb80765a154107550ea7$
 $3406400889f91ec6c5b143014c66aeb5c25fa23727f75338ad89e5a2e0667877f6dd5ab0d07ae88$
 $00ea9ff0c2679e070d27987b67472f3243e4075b74ce638cf2398a9e9e6524ba64bc61c9877bb75$
 $3e2b0135ad749f9fe168a257992afcdda866c9fca3ef5be332e3f0c48db180e5c0002232648eb75$
 $09b4b76deb768d4b10f53dfb094c7433a9fc654433d3a43e4abc98cb3272d04fac7a3062a0c82fe$
 $f8f47cd7e106e9e8ffe8f385d538a4b11c62a226f0aabdbfb1019e2d1dd62dbf14ad84603c78079$
 $ddf26107a59a0cf6ccfac97363$

$x_5 = 0x4838c54c0d30c0d28a00fa744250370a4880e4b057143a206b1355d194fa23bf3002cb35be$
 $42839fc4160ce5cc54098a675fea6e3e79845a0ee6e5c069bfe5ebf7745c2469efbc5b40fa0f594$
 $4dd24361e4df45fe8cd78652544baa6969eb12c95ac077240ff14eab69746758e616034e4b8c2d3$
 $392f1bd45f45f4b729cc4e10f402929168ee2632c5d0158a755a038c771bed43af44b7ff9574ae0$
 $0166e316db833ead6135ac287ba594c3eb803b6b5daee496df0c2535add8cb2467c1842f906e128$
 $0a63aa3b5395836d8578aee72c5d83e55c91d65705383f3ec24b43d021f37712973490af72adabe$
 $6a3eb96510676c7a0c5020726f61d7c4c2e81130833fafb70f7ede2d14a40a72779e731b27ac24f$
 $42942d71fc5c31fd88ea273c5b1d4675076af7325da12eb2b790ad44477f4e2ac8a03d5d18f146b$
 $803fb73bc0aa7454834e85c6a2dc058e96870a36020f156fb76fba7d75d08ce5f3b60d26fc2b8a6$
 $d09ac35452ef99dcd56e74520c7ceb33ac4c9e02a5e4e503f69dcb2b6fa788b2bdce99c72acaeb7$
 $ec41613b897fb73822bdf08f8b7ad6e5275bb762e8ce66465adba750b94054af4d23d8ee930563f$
 $bffcd44b373f7cb6b9515eb9cc57217313210bd8b7adf210be2f09ae0ddd03b98eed543d7603cef$
 $b2652f8ab81508148aa9459ae20baf045626d21176939011e1c25e101a6e55a3af8c2c95d3368f8$
 $f2995b1f5049dc7a494013327dbfffb698cfac377333efbfa3959330534947aa15ff850873a7a579$
 $6b711477855a171e0b1f11c3e32957a52c39260f42559b7f8ab49365cc91b055c97139a9d1531d9$
 $c79f053f041ada5faa2f43763d1d6f19f110f64a55ea90bf3d2140185f65204c52519a68853dbd9$
 $e60a263a6b41a8d72d65076b10b2a3f90a641b2d0827656d0a9cb9734a98d90cca114415aa07252$
 $fdbbe956dccf1d6ef3c37f6ef9$

Calculating X

To calculate X you use the following formula. $X = x_i + x_{i+1} \dots x_j \pmod{N}$

I'm not putting those giant numbers in here that'd take up too much ink.

Therefore

$X = 0xd49c77db966ba3079f5e264ab4969e3d231ba9b1357a3056f1ed427117764dfea3b1217f190$
 $203eb2afe989ce6b8eec4ae63bf366ee52082f8f1469a53af8fbc8224759878952dca98ac1c2e2a$
 $4df7fd0ade4f5c62b2e225eaa920b85dcf964cdeca516e5bdd65cc996c1de46b0d54abd8860e590$
 $fbcb0c8afb9c6f4bf0c506dd95f988291ff570d155aeac938785bce0823a5686d36eccf468cf3c16$
 $51d4e0da9b5e097d928e2aa4f39b07a79a6307b641101b3232f3c5ff39c70ff103c539c1d91a275$
 $31d2de133f783a681de9160146734755ead88e5804aec6e76ba45a425571fce79d789b18fc61631$
 $4f50b23badc585b6d0c109d326a7fcdb14d7d6ad8463b78b943fadbd1df0b8730b80642628e48e1c$
 $8cb0ecd4712c0b10b2b823cb747940197603c105b69680e96fb87ca5d24d27e7f5953cf32b9ea46$
 $8a580c315a7a542bfcc545981be56c09a65c38c2f165e79bd2ac92c283493ed8231186e6d15f5326$
 $b37b75692efe9481af448ab5585b3c2fc51c386448aa3e3f81e1b4f702ab4a94393e0fcdb677636$
 $3286d26c1234a48628638f98973b1262b6edef1b8e19132832779d3f5aac78c0605c5dbb0fb1b6d$
 $7bd124eb5c8e2332917d50e7a5bd292bfa17d2f20910aef12f797f7e475b543e027afb3d3e0fab2$
 $a783b293dc88f6b358ca10c0e24ee0144903e9e67ef30c6ab92c29fe0bc208524ef2ade1b8aaeff$
 $9e4afc3a2ad52eb7ba346f394600a5ccfc8ec5367bd68b0e9759ab1842ac5c9a42db3531a407ea2$
 $53767ac7da6d893957fda620910f578500866e16cb46afd6876e9b8a1a5c2d27050b84e8d138988$

f15441fc875f80c48119709f1a2d489d98d422724aad7726efec13f7de07e89248decf81aff6de6
e4ab2d5cdd1df117a625f588bcabb700d56d431b7205cb84bacf33e8372f3a23c837ceb6e6a1705
6b90fec6933809bf5bec6a00b

To get back the value M you use the following formula.

$$M = \sqrt[e]{X}$$

Thus

$M=0x546865726520617265206e6f2043544673207574696c697a696e67204861737461642042726$
 $f61646361737420736f207468697320616c6c20796f75206765742e198453179751900963952793$
 $218786024520485805055475877824165953272477019515416986218030$

After decoding M we get.

m ="There are no CTFs utilizing Hastad Broadcast so this all you get."

Your flag is: There are no CTFs utilizing Hastad Braodcast so this is all you get.

Thus the attack is complete.

Gimmick Keys

Sometimes you will find yourself dealing with a "clever" person who decides to use a gimmick prime where the modulus is not itself made up of 2 primes but a single prime raised to the same value multiple times. So to show this "wonderful" thing that drove me crazy I'll share a variant of this that I recently had to deal with. Now the actual challenge itself had an additional step where you had to decrypt a file with the final decrypted message but for simplicity's sake I'm going to stop after decrypting the RSA encrypted message. Also they somehow were able to create a key from their values that were invalid which is beyond the scope of this paper. You can use openssl to get the modulus/private key exponent from the file yourself. Also be sure to checkout <http://factordb.com/> to see if your modulus is already there as it'll save you the time. Otherwise you'll have to try carrying out Nth roots of the modulus until you find one. Also we're going to use the standard RSA encoding to keep the primes smaller for the purpose of this paper but be aware that Radford's method does exist.

The Challenge

First assume you have intercepted the following information from an archive. There is a RSA public key, and a text document that is just hex characters. You try to do fermat's near prime factorization, and weiner's attack, and coppersmith's and none of them work thus meaning you have to factor it.

After decoding the key you find the following information.

Modulus: 5715aa1ecfd537ca06c53ddff47f03bba220c86087a034927ce1f0388527

Public Key Exponent: 5

The Message: 9cbe0bbb478307428f1fdb81216f1591a1c007cc096aa9bba8d8c72bb5d

Factorization

After decoding the modulus to decimal(as factordb doesn't work with hex-encoded primes). We go to factordb and search for

601036006815631613350909380596125361015962093399321766034537645398852903

And sure enough it's a gimmick prime. It's the prime 843917831716618440759287 raised to the third power.

So that means that $p = 843917831716618440759287$ and $q = p^3$. Or

$p=843917831716618440759287$

$q=712197306689278721873900945929265731481076748369$

Calculating the Decryption Key

Since we now know that the prime is a gimmick one we can't use euler's theorem for calculating it as n due to n being a single prime repeated three times.

So then we use carmicheal's totient function to calculate the $\lambda(n)$. Through the following formula

$$\lambda(n) = lcm(p-1, q) \quad \text{After carrying this out through the knowledge that } lcm(p, q) = \frac{p * q}{gcd(p, q)}$$

And since we know that q is just 2 primes, and the fact that p-1 is no longer a factor of q. We can simply carry out (p-1) * q to get $\lambda(n)$. Thus we now have $\lambda(n)$ as

601036006815631613350908668398818671737240219498375836768806164322104534. To review here are the values that we know.

```

λ(n) = 601036006815631613350908668398818671737240219498375836768806164322104534
n = 601036006815631613350909380596125361015962093399321766034537645398852903
e=5

```

So now we have calculate the private key exponent d . We do this with the `mod_inv` function in the source code section. Like so.

```
d=mod_inv(e,λ(n))
```

We get d as 120207201363126322670181733679763734347448043899675167353761232864420907. With this information we can then decrypt the message.

Recall that RSA is based upon the basic mathematical formula for decryption. $P' = Ct^d \pmod{n}$

Where P' is the integer representation of the Plain-text message. For this message we're going to use the RSA standard OS2IP and I2OSP function named "rsa_ascii_encode" and "rsa_ascii_decode" respectively in the source code.

So then we then get the following formula we're going to use. For reduction of the number of pages used the formulas below will be using the hex version of the numbers. Below I'm going to use the "pow" function to represent it. And also "%" represents mod.

```

d = 0x116abb9fc32aa4c2015a8d06627942c84aca175132505ccadc79194c242b
Ct = 0x9cbe0bbb478307428f1fdb81216f1591a1c007cc096aa9bba8d8c72bb5d
n = 0x5715aa1ecfd537ca06c53ddff47f03bba220c86087a034927ce1f0388527
M' =
pow(0x9cbe0bbb478307428f1fdb81216f1591a1c007cc096aa9bba8d8c72bb5d, 0x116abb9fc32
aa4c2015a8d06627942c84aca175132505ccadc79194c242b) %
0x5715aa1ecfd537ca06c53ddff47f03bba220c86087a034927ce1f0388527
M' = 6889901333430938079884008768052087888977612466350811791270660104750

```

Then after converting it we get the secret message. "All Gimmick Primes are dumb."