20:09 RSA GTFO

by Ben Perez

I'd like to start off by saying: "Fuck RSA." Fuck the company RSA, fuck the conference, and fuck these things:

To properly motivate why I have these feelings about RSA, I'm going to have to introduce some mathematical foundations. RSA was invented as a result of a night of drinking "liberal quantities of Manischewitz wine"[39] in 1977, which was the same year Elvis died. If you encode "Rivest," "Shamir," "Adelman," and "Elvis" using the Chaldean numerology system and take their sum,

$$\begin{tikzcd}
\textsc{Rivest} & \textsc{Shamir} & & \textsc{Adelman} & \textsc{Elvis} \\
21 \arrow[drr] & 16 \arrow[dr] & & 23 \arrow[dl] & 18 \arrow[dll] \\
& & 78 & &
\end{tikzcd}$$

the result is 78. Adding the proper RSA key size in 2019, and subtracting the number of days Barack Obama was president,

$$78 + 4096 - 2920,$$

we arrive at 1254, the year in which the Catholic church created the dogma surrounding purgatory. Finally, divide this value by the number of felonies to which Jeffrey Epstein pled guilty before he was murdered, and add Buzz Aldrin's age when he faked the moon landing:

$$1254 \div 2 + 39 = 666.$$

That's right: Mathematical proof that RSA is the devil's work. □

But if pure logic won't convince you, perhaps we could take a look at how RSA actually works.

What is RSA again?

RSA is a public-key cryptosystem that has two primary use cases. The first is public key encryption, which lets a user, Alice, publish a public key that allows anyone to send her an encrypted message. The second use case is digital signatures, which allow Alice to "sign" a message so that anyone can verify the message hasn't been tampered with. The convenient thing about RSA is that the signing algorithm is basically just the encryption algorithm run in reverse. Therefore for the rest of this post we'll often refer to both as just RSA.

To set up RSA, Alice needs to choose two primes p and q that will generate the group of integers modulo $N = pq$. She then needs to choose a public exponent e and private exponent d such that $ed = 1 \bmod (p-1)(q-1)$. Basically, e and d need to be inverses of each other.

Once these parameters have been chosen, another user, Bob, can send Alice a message M by computing $C = M^e (\bmod N)$. Alice can then decrypt the ciphertext by computing $M = C^d (\bmod N)$. Conversely, if Alice wants to sign a message M, she computes $S = M^d (\bmod N)$, which any user can verify was signed by her by checking $M = S^e (\bmod N)$.

That's the basic idea. We'll get to padding–essential for both use cases–in a bit, but first let's see why, during every step of this process, things can go catastrophically wrong.

Revolving
Dating Stamp
SAMPLE POST-PAID FOR
50 cents.
THREE FOR A DOLLAR.
All the Months and Years from 1895 to 1900, Figures 0 to 99, "Rec'd," "Ans'd," "Paid," "Ac'p'd," "Ent'd."
HALF ACTUAL SIZE.
SIZE OF TYPE:
DEC 25 1895
D. T. MALLETT,
Broadway and Chambers Street, - New York City
FRONT

[39] *The RSA Cryptosystem: History, Algorithm, Primes*, 2007, by Michael Calderbank. `unzip pocorgtfo20.pdf historyofrsa.pdf`

Setting Yourself Up for Failure

RSA requires developers to choose quite a few parameters during setup. Unfortunately, seemingly innocent parameter-selection methods degrade security in subtle ways. Let's walk through each parameter choice and see what nasty surprises await those who choose poorly.

Prime Selection

RSA's security is based off the fact that, given a (large) number N that's the product of two primes p and q, factoring N is hard for people who don't know p and q. Developers are responsible for choosing the primes that make up the RSA modulus. This process is extremely slow compared to key generation for other cryptographic protocols, where simply choosing some random bytes is sufficient. Therefore, instead of generating a truly random prime number, developers often attempt to generate one of a specific form. This almost always ends badly.

There are many ways to choose primes in such a way that factoring N is easy. For example, p and q must be globally unique. If p or q ever gets reused in another RSA moduli, then both can be easily factored using the GCD algorithm. Bad random number generators make this scenario somewhat common, and research has shown that roughly one percent of TLS traffic in 2012 was susceptible to such an attack.[40] Moreover, p and q must be chosen independently. If p and q share approximately half of their upper bits, then N can be factored using Fermat's factorization method. In fact, even the choice of primality testing algorithm can have security implications.[41]

Perhaps the most widely-publicized prime selection attack is the ROCA vulnerability in RSALib which affected many smartcards, trusted platform modules, and even Yubikeys. Here, key generation only used primes of a specific form to speed up computation time. Primes generated this way are trivial to detect using clever number theory tricks. Once a weak system has been recognized, the special algebraic properties of the primes allow an attacker to use Coppersmith's method to factor N. More concretely, that means if the person sitting next to me at work uses a smartcard granting them access to private documents, and they leave it on their desk during lunch, I can clone the smartcard and give myself access to all their sensitive files.

It's important to recognize that in none of these cases is it intuitively obvious that generating primes in such a way leads to complete system failure. Really subtle number-theoretic properties of primes have a substantial effect on the security of RSA. To expect the average developer to navigate this mathematical minefield severely undermines RSA's safety.

Private Exponent

Since using a large private key negatively affects decryption and signing time, developers have an incentive to choose a small private exponent d, especially in low-power settings like smartcards. However, it is possible for an attacker to recover the private key when d is less than the 4th root of N. Instead, developers are encouraged to choose a large d such that Chinese remainder theorem techniques can be used to speed up decryption. However, this approach's complexity increases the probability of subtle implementation errors, which can lead to key recovery. In fact, last Summer Aditi Gupta modeled this class of vulnerabilities with the symbolic execution tool Manticore.[42]

People might call me out here and point out that normally when setting up RSA you first generate a

[40]`unzip pocorgtfo20.pdf weakkeys12.pdf`
[41]`unzip pocorgtfo20.pdf primeandprejudice.pdf`
[42]`https://blog.trailofbits.com/2018/08/14/fault-analysis-on-rsa-signing/`

modulus, use a fixed public exponent, and then solve for the private exponent. This prevents low private exponent attacks because if you always use one of the recommended public exponents (discussed in the next section) then you'll never wind up with a small private exponent. Unfortunately this assumes developers actually do that. In circumstances where people implement their own RSA, all bets are off in terms of using standard RSA setup procedures, and developers will frequently do strange things like choose the private exponent first and then solve for the public exponent.

Public Exponent

Just as in the private exponent case, implementers want to use small public exponents to save on encryption and verification time. It is common to use Fermat primes in this context, in particular $e = 3$, 17, and 65537. Despite cryptographers recommending the use of 65537, developers often choose $e = 3$ which introduces many vulnerabilities into the RSA cryptosystem.

When $e = 3$, or a similarly small number, many things can go wrong. Low public exponents often combine with other common mistakes to either allow an attacker to decrypt specific ciphertexts or factor N. For instance, the Franklin-Reiter attack allows a malicious party to decrypt two messages that are related by a known, fixed distance. In other words, suppose Alice only sends "chocolate" or "vanilla" to Bob. These messages will be related by a known value and allow an attacker Eve to determine which are "chocolate" and which are "vanilla." Some low public exponent attacks even lead to key recovery. If the public exponent is small (not just 3), an attacker who knows several bits of the secret key can recover the remaining bits and break the cryptosystem. While many of these $e = 3$ attacks on RSA encryption are mitigated by padding, developers who implement their own RSA fail to use padding at an alarmingly high rate.

RSA signatures are equally brittle in the presence of low public exponents. In 2006, Bleichenbacher found an attack which allows attackers to forge arbitrary signatures in many RSA implementations, including the ones used by Firefox and Chrome.[43] This means that any TLS certificate from a vulnerable implementation could be forged. This attack takes advantage of the fact that many libraries use a small public exponent and omit a simple padding verification check when processing RSA signatures. Bleichenbacher's signature forgery attack is so simple that it is a commonly used exercise in cryptography courses.[44]

Parameter Selection is Hard

The common denominator in all of these parameter attacks is that the domain of possible parameter choices is much larger than that of secure parameter choices. Developers are expected to navigate this fraught selection process on their own, since all but the public exponent must be generated privately. There are no easy ways to check that the parameters are secure; instead developers need a depth of mathematical knowledge that shouldn't be expected of non-cryptographers. While using RSA with padding may save you in the presence of bad parameters, many people still choose to use broken padding or no padding at all.

Padding Oracle Attacks, Everywhere

As we mentioned above, just using RSA out of the box doesn't quite work. For example, the RSA scheme laid out in the introduction would produce identical ciphertexts if the same plaintext were ever encrypted more than once. This is a problem, because it would allow an adversary to infer the contents of the message from context without being able to decrypt it. This is why we need to pad messages with some random bytes. Unfortunately, the most widely used padding scheme, PKCS #1 v1.5, is often vulnerable to something called a padding oracle attack.

Padding oracles are pretty complex, but the high-level idea is that adding padding to a message requires the recipient to perform an additional check: whether the message is properly padded. When the check fails, the server throws an invalid padding error. That single piece of information is enough to slowly decrypt a chosen message. The process is tedious and involves manipulating the target ciphertext millions of times to isolate the changes which result in valid padding. But that one error message is all you need to eventually decrypt a chosen ciphertext. These vulnerabilities are particularly bad because attackers can use them to recover

[43] `https://www.imperialviolet.org/2014/09/26/pkcs1.html`
[44] `https://cryptopals.com/sets/6/challenges/42`

pre-master secrets for TLS sessions. For more details on the attack, there is an excellent explainer on StackExchange.[45]

The original attack on PKCS #1 v1.5 was discovered way back in 1998 by Daniel Bleichenbacher. Despite being over 20 years old, this attack continues to plague many real-world systems today. Modern versions of this attack often involve a padding oracle slightly more complex than the one originally described by Bleichenbacher, such as server response time or performing some sort of protocol downgrade in TLS. One particularly shocking example was the ROBOT attack, which was so bad that a team of researchers were able to sign messages with Facebook's and PayPal's secret keys. Some might argue that this isn't actually RSA's fault—the underlying math is fine, people just messed up an important standard several decades ago. The thing is, we've had a standardized padding scheme with a rigorous security proof, OAEP, since 1998. But almost no one uses it. Even when they do, OAEP is notoriously difficult to implement and often is vulnerable to Manger's attack, which is another padding oracle attack that can be used to recover plaintext.

The fundamental issue here is that padding is necessary when using RSA, and this added complexity opens the cryptosystem up to a large attack surface. The fact that a single bit of information, whether the message was padded correctly, can have such a large impact on security makes developing secure libraries almost impossible. TLS 1.3 no longer supports RSA so we can expect to see fewer of these attacks going forward, but as long as developers continue to use RSA in their own applications there will be padding oracle attacks.

COAL HANDLING MACHINERY
FOR
Coal Mines—Coal Yards—Boiler Rooms
Electric Coal Cutters, Drills, Locomotives, Ventilating Fans, Mine Hoists, Etc., Wood or Steel Tipples and Tipple Equipment.
Car Hauls, Picking Tables, Screens, Crushers, Pulverizers, Etc. Elevators and Conveyors for Handling Materials of all kinds.
The Jeffrey Mfg. Company
COLUMBUS, OHIO, U. S. A.

So what should you use instead

People often prefer using RSA because they believe it's conceptually simpler than the somewhat confusing DSA protocol or moon math elliptic curve cryptography (ECC). But while it may be easier to understand RSA intuitively, it lacks the misuse resistance of these other more complex systems.

First of all, a common misconception is that ECC is super dangerous because choosing a bad curve can totally sink you. While it is true that curve choice has a major impact on security, one benefit of using ECC is that parameter selection can be done publicly. Cryptographers make all the difficult parameter choices so that developers just need to generate random bytes of data to use as keys and nonces. Developers could theoretically build an ECC implementation with terrible parameters and fail to check for things like invalid curve points, but they tend to not do this. A likely explanation is that the math behind ECC is so complicated that very few people feel confident enough to actually implement it. In other words, it intimidates people into using libraries built by cryptographers who know what they're doing. RSA on the other hand is so simple that it can be (poorly) implemented in an hour.

Second, any Diffie-Hellman based key agreement or signature scheme (including elliptic curve variants) does not require padding and therefore completely sidesteps padding oracle attacks. This is a

[45] `https://crypto.stackexchange.com/questions/12688/can-you-explain-bleichenbachers-cca-attack-on-pkcs1-v1-5`

major win considering RSA has had a very poor track record avoiding this class of vulnerabilities.

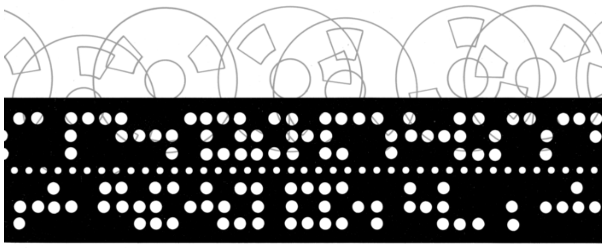
We recommend using Curve25519 for key exchange and digital signatures. Encryption needs to be done using a protocol called ECIES which combines an elliptic curve key exchange with a symmetric encryption algorithm. Curve25519 was designed to entirely prevent some of the things that can go wrong with other curves, and is very performant. Even better, it is implemented in libsodium, which has easy-to-read documentation and is available for most languages.

Seriously, stop using RSA

RSA was an important milestone in the development of secure communications, but the last two decades of cryptographic research have rendered it obsolete. Elliptic curve algorithms for both key exchange and digital signatures were standardized back in 2005 and have since been integrated into intuitive and misuse-resistant libraries like libsodium. The fact that RSA is still in widespread use today indicates both a failure on the part of cryptographers for not adequately articulating the risks inherent in RSA, and also on the part of developers for overestimating their ability to deploy it successfully.

The security community needs to start thinking about this as a herd-immunity problem—while some of us might be able to navigate the extraordinarily dangerous process of setting up or implementing RSA, the exceptions signal to developers that it is in some way still advisable to use RSA. Despite the many caveats and warnings on StackExchange and Github READMEs, very few people believe that they are the ones who will mess up RSA, and so they proceed with reckless abandon. Ultimately, users will pay for this. This is why we all need to agree that it is flat out unacceptable to use RSA in 2019. No exceptions.

Fuck RSA.



All About

OSI

BASIC-IN-ROM

Ohio Scientific Microsoft BASIC Ver 1.0 Rev 3.2
REFERENCE MANUAL

Complete, Concise, Accurate, Detailed. All commands, statements, and functions. Maps. USR. Tapes. Bug fixes. Variable tables. Source code storage. MONITOR.

Postpaid $8.95 Send check, or COD

EDWARD H. CARLSON ✓259
3872 RALEIGH DR.
OKEMOS, MI 48864

Dealer Inquiries Welcome

CANADIANS!

Eliminate the Customs Hassles. Save Money and get Canadian Warranties on IMSAI and S-100 compatible products.

IMSAI 8080	KIT	$ 838.00
	ASS.	$1163.00

(Can. Duty & Fed. Tax Included).

AUTHORIZED DEALER

Send $1.00 for complete IMSAI Catalog.

We will develop complete application systems.

Contact us for further information.

Rotundra Cybernetics

Box 1448, Calgary, Alta. T2P 2H9
Phone (403) 283-8076