Base64 Malleability in Practice

Panagiotis Chatzigiannis[1] and Konstantinos Chalkias[2]

[1] George Mason University
pchatzig@gmu.edu
[2] Meta
kostascrypto@fb.com

Abstract. Base64 encoding has been a popular method to encode binary data into printable ASCII characters. It is commonly used in several serialization protocols, web, and logging applications, while it is oftentimes the preferred method for human-readable database fields. However, while convenient and with a better compression rate than hex-encoding, the large number of base64 variants in related standards and proposed padding-mode optionality have been proven problematic in terms of security and cross-platform compatibility. This paper addresses a potential attack vector in the base64 decoding phase, where multiple different encodings can successfully decode into the same data, effectively breaking string uniqueness guarantees. The latter might result to log mismatches, denial of service attacks and duplicated database entries, among the others. Apart from documenting why canonicity can be broken by a malleable encoder, we also present an unexpected result, where most of today's base64 decoder libraries are not 100% compatible in their default settings. Some surprising results include the non-compatible behavior of major Rust base64 crates and between popular Javascript and NodeJS base64 implementations. Finally, we propose ways and test vectors for mitigating these issues until a more permanent solution is widely adopted.

Keywords: base64, malleability, encoding incompatibility, padding attacks

1 Introduction

1.1 Base64 encoding description

Base64 encoding offers a way to represent binary data in readable format using printable ASCII characters. Such representation is particularly common for exchanging binary data over emails and web pages, and is the recommended way by the developer community to add binary data into JSON data structures [15]. For instance, base64 encoded data (e.g. images) can be embedded inline in text documents [5]. Note that this encoding is not an encryption scheme, which is a common misconception in the developer community [6].

Essentially, base64 encoding divides a group of 24-bit binary data into four 6-bit chunks, where each chunk is mapped to a printable ASCII character, as shown in Table 1. However, as the encoded binary data might not always be a multiple of 24, special padding characters (=) is usually used, resulting in a total of 64 + 1 unique characters used in base64 representations.

Base64 is encountered in several application-specific variations [11,7,8,4,1], with RFC4648 [9] commonly referenced as its standard. However, this plethora of available variations exacerbates the problems we discuss later in this paper.

Similarly to base64, base58 [12] was introduced to further improve readability by removing similarly-looking characters (e.g. zero "0" with capital "o", lowercase "L" with capital "i" etc.) as well as a few alphanumeric characters. However, the removal of 6 characters slightly reduced compression, while due to the fact that the 58 is not a power of 2, parsing is considered to be about 2% slower. The primary application of base58 is binary-to-text address encoding in Bitcoin [13], but still base64 is the preferred method for compressed human readable string representation of bytes in most other applications, when better than hexadecimal (base16) compression is required.

However, as we realized in practice, base64 decoder implementations are inconsistent across various systems, programming languages and software libraries. Another major reason causing confusion is the extensive use of expressions like 'one MAY ignore the pad character' and similar "optionality" features in the related standardization documents [9]. Unfortunately, this is an observed pattern in the majority of RFCs across multiple domains and along with non-existent public test vectors covering exploitable edge cases, many serious security and incompatibility issues have been recently reported in cryptography related standards [3]. In this paper, we identify such inconsistencies in some of the most popular base64 implementations, highlight potential real-world attacks due to implementation incompatibilities and canonicity misconceptions; and finally propose appropriate mitigations and test vectors to compare library behaviors.

Table 1. Base64 mapping table

Binary Data	Char	Binary Data	Char	Binary Data	Char	Binary Data	Char
000000	A	010000	Q	100000	g	110000	w
000001	B	010001	R	100001	h	110001	x
000010	C	010010	S	100010	i	110010	y
000011	D	010011	T	100011	j	110011	z
000100	E	010100	U	100100	k	110100	0
000101	F	010101	V	100101	l	110101	1
000110	G	010110	W	100110	m	110110	2
000111	H	010111	X	100111	n	110111	3
001000	I	011000	Y	101000	o	111000	4
001001	J	011001	Z	101001	p	111001	5
001010	K	011010	a	101010	q	111010	6
001011	L	011011	b	101011	r	111011	7
001100	M	011100	c	101100	s	111100	8
001101	N	011101	d	101101	t	111101	9
001110	O	011110	e	101110	u	111110	+
001111	P	011111	f	101111	v	111111	/

1.2 Related works

We now review a number of related works on base64. Concurrently with our work, a recent whitepaper from SANS Institute [6] highlights several potential security issues associated with using base64. These issues include developer misconceptions that base64 offers cryptographic encryption rather than encoding, exposed passwords in web-based base64 authentication, embedding Javascript in URLs and cross-site scripting. Another potential problem with base64 is that it can potentially facilitate data leakage from an organization (since base64 data format might not be immediately detected), or even harming intellectual rights, e.g. by prepending spaces in binary data which would result in a totally different base64 encoding. Also in an interesting twist, although base64 itself is not an encryption scheme, [19] considered using base64 encodings in conjunction with Caesar cipher for securing video files. Also, [17] discusses how to expose naive attempts to use base64 encodings as an encryption method.

Works more broadly related to base64 encoding include base64 implementation in PHP [18], a variation of base64 for encoding URLs [16], using base64 as a compression method when used along with encryption algorithms such as AES to improve their efficiency [14] and in X.509 digital certificates and cryptographic keys PEM formats [10].

2 Inconsistencies and Attacks

In the previous section we discussed how the absence of canonicity in base64 decoding results in inconsistent implementations. Briefly, although correct base64 encoding implementations will always produce consistent and unique string results, it is possible for multiple base64 decoders to either ignore padding bits completely or even omit checking whether padding was applied correctly. It is also highlighted that apart from the "=" padding symbol(s) at the end of (some) base64 strings, because each base64 character represents 6-bits of information, any binary input whose length is not a multiple of 6, is padded with zero bits, until it gets 6-bit aligned. For instance, the 32-bit hex value `0x433356c1` normally encodes into `QzNWwQ==`; note that 32 is not divided by 6 and a correct encoder adds 4 zero bits at the end of the binary to make it 6-bit aligned.

However, a base64 decoder decodes into actual bytes (8-bit aligned). An observed issue in our experiments is that padding truncation happens blindly in some implementations and they do not check if the padding bits were all zeros. For instance, the following strings:

`QzNWwQ==` (`010000 110011 001101 010110 110000` **`010000`** in binary)
`QzNWwc==` (`010000 110011 001101 010110 110000` **`011100`** in binary)

will be successfully decoded to the same data if the last 4 zero padding bits check gets omitted. Obviously, in the above example `QzNWwc==` is padded incorrectly as the last 4 bits of the last 6 bit chunk should be zero.

In Table 2 we show a number of tests we performed for a variety of programming or scripting languages and systems. We specifically used the binary

Table 2. Base64 decodings on test vectors

Test vector	`SGVsbG8=`	`SGVsbG9=`	`SGVsbG9`	`SGVsbA==`	`SGVsbA=`	`SGVsbA`	`SGVsbA====`
PHP	`Hello`	`Hello`	`Hello`	`Hell`	`Hell`	`Hell`	`Hell`
PHP (strict)	`Hello`	`Hello`	`Hello`	`Hell`	-	`Hell`	-
Go	`Hello`	`Hello`	illegal base64 data	`Hell`	illegal base64 data	illegal base64 data	illegal base64 data
Haskell (base64-bytestring)	`Hello`	non-canonical encoding detected	bytestring is unpadded or has invalid padding	`Hell`	bytestring is unpadded or has invalid padding	bytestring is unpadded or has invalid padding	bytestring is unpadded or has invalid padding
Ruby	`Hello`	`Hello`	`Hello`	`Hell`	`Hell`	`Hell`	-
JS	`Hello`	`Hello`	`Hello`	`Hell`	exception	`Hell`	exception
nodejs	`Hello`	`Hello`	`Hello`	`Hell`	`Hell`	`Hell`	`Hell`
C#	`Hello`	`Hello`	Invalid length	`Hell`	Invalid length	Invalid length	not a valid Base-64 string
OCaml	`Hello`	`Hello`	Wrong padding	`Hell`	Wrong padding	Wrong padding	Too much input
Perl	`Hello`	`Hello`	`Hello`	`Hell`	`Hell`	`Hell`	`Hell`
R	`Hello`	`Hello`	`Hello`	`Hell`	`Hell`	`Hell`	`Hell`
OpenSSL	`Hello`	`Hello`	-	`Hell`	-	-	-
MySQL	`Hello`	`Hello`	`NULL`	`Hell`	`Hell`	`Hell`	-
Windows PS	`Hello`	`Hello`	Invalid length	`Hell`	Invalid length	Invalid length	not a valid Base-64 string
Unix	`Hello`	`Hello`	invalid input	`Hell`	invalid input	invalid input	invalid input
Python	`Hello`	`Hello`	incorrect padding	`Hell`	incorrect padding	incorrect padding	`Hell`
Rust (crate base64 0.1.0)	`Hello`	`Hello`	`Hello`	`Hell`	`Hell`	`Hell`	`Hell`
Rust (crate base64 0.13.0)	`Hello`	Invalid Last Symbol	Invalid Last Symbol	`Hell`	`Hell`	`Hell`	Invalid byte
Rust (crate base64ct 1.3.3)	`Hello`	Invalid Encoding	Invalid Encoding	`Hell`	Invalid Encoding	Invalid Encoding	Invalid Encoding

representations of the words ‘Hello’ and ‘Hell’ as test vectors, as they normally require different number of padding “=” symbols and we could cover different padding handling combinations, including incorrectly non-zeroized padding bits and omitted or exceeded “=” symbols.

From this Table, we show that it is possible to slightly alter the base64-encoded data and still achieve the same decoded output. This can be an important attack vector in some languages used in back-end web development such as PHP, and our tests show that default base64 decoding logic does not handle any padding inconsistencies. In the opposite, some languages and libraries are quite resistant and reject most malleable-ized base64 encoded inputs. For instance, Haskell and Rust’s base64ct crate v.1.3.3 decoding implementations throw appropriate errors in all of our malleable-ized test vectors. We also note as interesting observations the different behavior between NodeJS and JS, as well as the different behavior among different libraries or even different version updates of the same library, as highlighted in the case of Rust.

As a result, a "malicious" base64 implementation could potentially encode the same data in a different way and alter some of the last characters of the original base64 output, without the application realizing the difference. For instance, one could potentially use a malicious base64 encoder in web-based ticketing applications, and buy multiple tickets for free if the database uses base64 string for unique ticketIDs [2]. In fact, many developers prefer human-readable unique IDs (for monitoring and logging purposes) and base64 is usually the preferred method, unfortunately under the assumption that base64 is always canonical. In short, in databases where their base64-encoded userIDs are exposed, an attacker (or internal actor) can potentially read a userID and insert multiple copies of the same user by just slightly altering padding bits, and thus bypassing the logical database's primary key uniqueness rules. In an another example, suppose there is a custom digital certificate repository, where certificates are validated for uniqueness based on their base64-encoded binary string, and because community utilizes PEM formats [10], one could alter the certification's base64 string representation. Using such attacks, one can re-register the same certificate. Another possible attack is intentionally bypassing checks against logs: altering a keyword's base64 encoding would no longer match that keyword against that log, effectively bypassing any idempotency checks.

No canonicity in base64 decoding implies another denial of service attack vector: one could replace a "=" padding character with many "=" (as shown in one of the test vectors of Table 2), possibly making a huge (e.g. several Gigabyte) base64 string being accepted, which would still decode into the same just a few bits data. If the size-check happens on the decoded output, the service might store an inappropriate amount of data.

Similarly, base64 non-canonicity might affect other serialization standards and proposals as well. An example is canonical JSON, where it is a common practice that binary data is encoded to base64 first. That would imply that malicious users could send two or more different JSONs representing the exact same object/data ad thus bypassing some of the rules, such as JSON hash for the same object being inconsistent. Along the same lines, if one is interested in non-malleable signatures (i.e., in some blockchains), ideally a provably canonical serialization engine should be used. All in all, the community should be better educated to design systems secure against malicious encoders when canonicity is an important feature.

To sum up, our test vectors show that many programming languages and systems are particularly susceptible to decoding malleability attacks. Another interesting finding is Python's default relaxed behavior of completely ignoring invalid base64 characters during decoding (e.g., both `SGVsbA====` and `SGVsb<A=>===` decode into `Hell`, since `>` and `<` are not part of base64 character set); a feature which if not properly addressed in Python applications, could cause further malleability issues. It is actually interesting that many implementations have an api for `strict` decoding, which unfortunately does not refer to canonicity, but rather it only checks for non-base64 characters. On the other hand, as previously discussed, some languages and libraries (e.g. Haskell and Rust's `base64ct` crate) are particularly sensitive, and thus indicate a more secure implementation.

3 Mitigation proposals

From the previous section we saw that base64 decoding malleability, where many different base64-encoded strings can decode into the same binary data, is an attack vector that has not been addressed properly so far by the community, thus highlighting the importance of canonicity or at least cross-platform compatibility. Therefore, in general, developers should never assume that there is an unique match between a binary input and its base64 representation, as multiple base64 encoded strings may represent the same value, and should never rely on a unique "1-1" property to secure their system. For instance, a database administrator should not use base64-encoded strings as primary keys, especially when those strings are received from external users. Still developers should prefer utilizing a more "malleability-resistant" library if it exists (e.g. prefer base64ct over base64 library in Rust).

Also, we observe the most popular RFC4648 standard for base64 leaves a large room for different implementation variations [9]. For instance, it defines the use of padding as optional (paragraph 3.2), providing the option to ignore the padding character or the excess padding during decoding (paragraph 3.3), and giving decoders the option to accept or reject inputs if the pad bits have not been properly set to zero (paragraph 3.5). Not mandating a specific behavior while providing such a wide range of implementation choices is problematic and we consider it as a bad practice [3]. Therefore, we recommend RFC standards to follow a stricter set of rules (or if optionality is really required, we should encourage standardizing different modes as well), while also including a set of failed test vectors and potential edge cases.

Another potential mitigation method, specifically when a canonical implementation is not available, would be to perform an additional check: re-encode the received decoded binary data, then check byte equivalence with the original input base64; if they do not match, then this indicates a malleability scenario. However, this additional cycle might be particularly costly in certain scenarios.

In a nutshell, a more permanent mitigation strategy would be to encourage (or enforce) strict and canonical checking in decoding implementations, by validating padding bits as well. Although this might have a slight negative impact on efficiency, the security benefits would be much more important by addressing the root of the problem.

4 Conclusions

In this paper we performed a short survey on various base64 decoding implementations, and showed how different programming languages and systems have colliding decoded outputs for different base64 encoded strings, or detect anomalies and throw exceptions, thus minimizing the attack surface. We should highlight that similar types of potential malleability attacks might exist in other data representation methods as well, e.g. when representing hexadecimal values with uppercase, lowercase or a mix of uppercase and lowercase letters.

We also believe that RFC standards should be written in a strict and precise format, without leaving room for "options" and avoiding words like "may or optionally" [9]. Our suggestion is to use different identifiers for separate standard if these various different ways of implementation need to be maintained.

Finally, we hope our paper will raise awareness of the associated security implications in the short term, as well as serve as a pathway towards permanently addressing those issues in the long term, by enforcing correct validation of padding bits across all languages and systems.

5 Acknowledgements

We would like to thank Jack Moffit and Rajath Shanbag for their implementation contribution regarding test vector compatibility in multiple requested programming languages, and Levi Schuck for precious feedback and review comments on late manuscript.

References

1. J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. Ietf rfc: 4880. `https://datatracker.ietf.org/doc/html/rfc4880`, 2007.
2. Konstantinos Chalkias. 3rd time that i find a serious exploitable bug in a multi-million business website. `https://www.linkedin.com/posts/chalkiaskostas_programming-python-databases-activity-6820220132233748480-8teK`, 2021.
3. Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many eddsas. In Thyla van der Merwe, Chris Mitchell, and Maryam Mehrnezhad, editors, *Security Standardisation Research*, pages 67–90, Cham, 2020. Springer International Publishing.
4. M. Crispin. Ietf rfc: 3501. `https://datatracker.ietf.org/doc/html/rfc3501#section-5.1.3`, 2003.
5. developer.mozilla.org. Data urls. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs`, 2022.
6. Kevin Fiscus. Base64 can get you pwned. `https://www.sans.org/white-papers/33759`, 2011.
7. N. Freed and N. Borenstein. Ietf rfc: 2045. `https://datatracker.ietf.org/doc/html/rfc2045#page-24`, 1996.
8. D. Goldsmith and M. Davis. Ietf rfc: 2152. `https://datatracker.ietf.org/doc/html/rfc2152`, 1997.
9. S. Josefsson. Ietf rfc: 4648. `https://datatracker.ietf.org/doc/html/rfc4648#section-3.5`, 2006.
10. Simon Josefsson and Sean Leonard. Textual encodings of pkix, pkcs, and cms structures. *Internet Engineering Task Force April*, 2015.
11. J. Linn. Ietf rfc: 1421. `https://datatracker.ietf.org/doc/html/rfc1421`, 1993.
12. S. Nakamoto and M. Sporny. The base58 encoding scheme. `https://tools.ietf.org/id/draft-msporny-base58-01.html`, 2019.
13. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2009.

14. Ajeet Ram Pathak, Sarita Deshpande, and Mudra Panchal. A secure framework for file encryption using base64 encoding. In Sheng-Lung Peng, Nilanjan Dey, and Mahesh Bundele, editors, *Computing and Network Sustainability*, pages 359–366, Singapore, 2019. Springer Singapore.
15. Amin Sh. Put byte array to json and vice versa. `https://stackoverflow.com/questions/20706783/put-byte-array-to-json-and-vice-versa`, 2013.
16. Prabath Siriwardena. *Base64 URL Encoding*, pages 397–399. Apress, Berkeley, CA, 2020.
17. Daniel Smallwood. Hacker pig latin: A base64 primer for security analysts. `https://www.darkreading.com/edge-articles/hacker-pig-latin-a-base64-primer-for-security-analysts`, 2021.
18. Wen Somchai and Dang Wen. Research on base64 encoding algorithm and PHP implementation. In Shixiong Hu, Xinyue Ye, Kun Yang, and Hongchao Fan, editors, *26th International Conference on Geoinformatics, Geoinformatics 2018, Kunming, China, June 28-30, 2018*, pages 1–5. IEEE, 2018.
19. Mars Caroline Wibowo, Phong Thanh Nguyen, Edmond Febrinicko Armay, and Robbi Rahim. Implementation of base64 and caesar cipher in securing video files. *Journal of Advanced Research in Dynamical and Control Systems*, 12(2):761–765, 2020.

A Test vector generation

Example library invocations for base64 decoders.

Ruby
```
require "base64"
Base64.decode64("SGVsbG8=")
```

NodeJS
```
Buffer.from("SGVsbG8=", "base64").toString()
```

C#
```
Convert.FromBase64String ("SGVsbG8=");
```

Php
```
print base64_decode("SGVsbG8=")
```

Haskell
```
Prelude Data.ByteString.Base64> decode "SGVsbG8="
```

Rust
```
let encoded = "SGVsbG8=";
let decoded = Base64::decode_vec(&encoded).unwrap();
```

Windows Powershell
```
[System.Text.Encoding]::ASCII.GetString ([System.Convert]::FromBase64String
("SGVsbG8="))
```

Unix
```
echo 'SGVsbG8=' | base64 --decode
```