ForgedAttributes: An Existential Forgery Vulnerability of CMS Signatures

Falko Strenzke

MTG AG, Darmstadt, Germany

Abstract. This work describes an existential signature forgery vulnerability of the current CMS signature standard. The vulnerability results from an ambiguity of how to process the signed message in the signature verification process. Specifically, the absence or presence of the so called SignedAttributes field determines whether the signature message digest receives as input the message directly or the SignedAttributes, a DER-encoded structure which contains a digest of the message. Two flavors of the attack are possible: For one, if an attacker takes a CMS signed message M which was originally signed with SignedAttributes present, then he can craft a new message M' that was never signed by the signer and has the DER-encoded SignedAttributes of the original message as its content and verifies correctly against the original signature of M, but here with SignedAttributes absent. The second attack is given by the inversion of the above one and thus requires the signer to sign a message in the specific form of an encoded SignedAttributes structure and to produce the signed message with SignedAttributes absent. This second variant leads to forged signed messages with arbitrary attacker-chosen content. We provide a proof-of-concept attack implementation, estimate the impact of the attack and discuss possible countermeasures.

E-mail: falko.strenzke@mtg.de (Falko Strenzke)

1 Introduction

The CMS [Hou09] standard and its predecessor PKCS#7 [Kal98] are cryptographic data formats that specify the format of encrypted and signed messages using X.509 certificates. Among other protocols, they are employed for instance in S/MIME [STR19] and protocols for certificate management [Gut20, LMS21]. In this work, we describe an existential signature forgery attack against the CMS and PKCS#7 protocols that is based on a design error in the standards. Namely, the signature verification has to be carried out in two different ways depending on the presence or absence of the optional SignedAttributes field in the SignerInfo structure of CMS signed messages (referred to as authenticatedAttributes in PKCS#7). The misleading notion here is that these attributes may seem to be protected through the signature, as, if they are present, their value is passed to the signature digest algorithm. However, the mechanism allows adding or removing the SignedAttributes and modifying the signed message accordingly to yield valid signatures for messages that were not intentionally signed by the legitimate signer. In this work, we present two variants of this attack that achieve existential signature forgeries and thus violate the security notion of Existential Unforgeability under Chosen Message Attacks (EUF-CMA) of the protocol.

2 CMS Signatures

CMS signatures are defined in [Hou09]. A signed message consists of a SignedData object with the following ASN.1 notation:

```
SignedData ::= SEQUENCE {
        version CMSVersion,
        digestAlgorithms DigestAlgorithmIdentifiers,
        encapContentInfo EncapsulatedContentInfo,
        certificates [0] IMPLICIT CertificateSet OPTIONAL,
        crls [1] IMPLICIT RevocationInfoChoices OPTIONAL,
        signerInfos SignerInfos }

SignerInfos ::= SET OF SignerInfo

EncapsulatedContentInfo ::= SEQUENCE {
        eContentType ContentType,
        eContent [0] EXPLICIT OCTET STRING OPTIONAL }
```

The SignerInfo contains the signature:

```
 SignerInfo ::= SEQUENCE {
        version CMSVersion,
        sid SignerIdentifier,
        digestAlgorithm DigestAlgorithmIdentifier,
        signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
        signatureAlgorithm SignatureAlgorithmIdentifier,
        signature SignatureValue,
        unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }

SignedAttributes ::= SET SIZE (1..MAX) OF Attribute


Attribute ::= SEQUENCE {
        attrType OBJECT IDENTIFIER,
```

```
        attrValues SET OF AttributeValue }

AttributeValue ::= ANY
```

The crucial point for the vulnerability is the optional presence of the SignedAttributes. If the content type specified in the ContentType field of the SignerInfo structure is "id-data", then the presence of the SignedAttributes in the SignerInfo is optional and their inclusion is at the discretion of the signer. For any other content-type, the SignedAttributes must always be present as per CMS specification. Note that the ContentType field itself is also not protected by the signature and thus its value can be manipulated by an attacker.

If the SignedAttributes are included, they must contain at least two specific attributes, namely the messageDigest and the contentType attribute which are identified by the following OIDs:

```
id-messageDigest OBJECT IDENTIFIER ::= { iso(1) member-body(2)
          us(840) rsadsi(113549) pkcs(1) pkcs9(9) 4 }

id-contentType OBJECT IDENTIFIER ::= { iso(1) member-body(2)
          us(840) rsadsi(113549) pkcs(1) pkcs9(9) 3 }
```

where the message digest attribute value is given by an OCTET STRING that contains the digest of the signed message as its value.

The CMS standard specifies the signature algorithm given in Algorithm 1. Here, the function invocation DER-encode(signedAttrs(M)) denotes the generation of the DER-encoded value of the SignedAttributes, which depends on M due to the mandatory messageDigest attribute which contains HASH(M). The function sign(K_s, D) computes the signature over the message digest D.

Algorithm 1 CMS signature generation algorithm.

1: **Algorithm** CMS-Sign(secret key K_s, message M)
2: **if** signedAttrs are absent **then**
3: $D = \text{HASH}(M)$
4: **else**
5: $\text{signedAttr}_M^{\text{DER}} = \text{DER-encode}(\text{signedAttrs}(M))$
6: $D = \text{HASH}(\text{signedAttr}_M^{\text{DER}})$
7: **end if**
8: return sign(K_s, D)
9: **end Algorithm**

The corresponding signature the verification algorithm is given in Algorithm 2.

3 PKCS#7 signatures

PKCS#7 signatures are defined in [Kal98]. They are mostly identical to CMS and regarding the features relevant to the vulnerability that is subject of this work, there is no difference between the two standards. The paragraphs describing the digest computation are almost literally identical between both standards. In the following, for the sake of simplicity, we will only refer to CMS signatures, while the attacks are generally also applicable to PKCS#7 signatures.

Algorithm 2 CMS signature verification algorithm. The parameter $\text{signedAttr}_M^{\text{DER}}$ denotes the DER-encoded SignedAttributes structure of the SignerInfo that is subject to verification and has the special value $\emptyset$ if the SignedAttributes are not present in the SignerInfo structure of the SignedData object.

Algorithm CMS-VERIFY(message M, public key K_p, $\text{signedAttr}_M^{\text{DER}}$, signature S)
  $D = \text{HASH}(M)$
  if $\text{signedAttr}_M^{\text{DER}} = \emptyset$ **then**
    $E = D$
  else
    $F =$ message digest for M contained in $\text{signedAttr}_M^{\text{DER}}$
    if $F \neq D$ **then**
      return FAIL
    end if
    $E = \text{HASH}(\text{signedAttr}_M^{\text{DER}})$
  end if
  return verify(K_p, E, S)
end Algorithm

4 Existential signature forgery attacks

In the following subsections, two variants of an existential forgery attack against CMS are described. Table 1 gives an overview of the two variants.

4.1 Variant 1: Attacker removes SignedAttributes

An existential forgery attack against CMS signatures, which we label as the "Remove-SA attack" variant, can be conducted as follows. The signer signs an arbitrary message M with SignedAttributes present. The attacker takes the resulting SignerInfo from the SignedData and removes the SignedAttributes from it. Then he replaces the content in the eContent field, i.e. the signed message, with the same encoded SignedAttributes[1] that he removed in the prior step. This means that the verifier processes the message according to SignedAttributes being absent and successfully verifies the signature for the message

$$M' = \text{signedAttr}_M^{\text{DER}} = \text{DER-encode}(\text{signedAttrs(M)})$$

that was never signed by the signer.

4.2 Variant 2: Attacker adds SignedAttributes

In the second variant, the "Add-SA attack" the signer signs an attacker-chosen message which has the form of a valid DER-encoded SignedAttributes structure. The signature generation is performed with SignedAttributes absent in the SignerInfo.

In this attack, the attacker crafts the signed data in the form of the SignedAttributes structure such that the message digest attribute contained in them is the hash of a message M' of his choice. After having received the signature for the intentionally signed message, the attacker exchanges the message with the message M' and adds the originally signed message as the SignedAttributes to the CMS SignerInfo structure. The verification succeeds since the signed message is processed according to the case where SignedAttributes are present in Algorithm 2.

[1] Note that there is a difference between the encoding of the first byte of the structure of the SignedAttributes depending on whether they are placed within a SignerInfo structure or stand for themselves. They are fed to the message digest in the latter form. See Appendix A for the details.

Table 1: Overview of the two attack variants. The first row gives the precondition that the SignerInfos structure of the legitimately signed message must fulfill for the respective attack variant. The second row describes the requirements that have to be fulfilled by the message that the legitimate signer needs to sign. The third row indicates the transformation that is applied to the CMS SignerInfo structure by the attacker. The forth row describes the form of the crafted message, for which the legitimate signature is also valid. The final two rows specify restrictions on the content-type in the original and the forged message.

	Remove-SA attack	**Add-SA attack**
Precondition for SignerInfo	SignedAttributes are present	SignedAttributes are absent
Form of the intentionally signed message	arbitrary	valid DER-encoded SignedAttributes chosen by the attacker
Transformation applied to SignerInfo	removal of SignedAttributes from SignerInfo	addition of SignedAttributes to SignerInfo
Form of the forged message	valid DER-encoded SignedAttributes as produced by the signer	arbitrarily chosen by the attacker
Content-type of original message	any	= id-data
Content-type of forged message	= id-data	any

5 Proof-of-concept implementation of the Remove-SA Attack

We created an attack tool that executes the Remove-SA attack against CMS SignedData files and was tested successfully against OpenSSL. The attack is simulated with the following command sequence:

```
openssl cms -sign -nodetach -signer signcert.pem -inkey signcert_pri.pem \
-binary -in data.file -outform DER -out test_sign.der

./forged_attributes.py test_sign.der

openssl cms -verify -signer signcert.pem  -in test_sign.der \
-inform DER  -noverify

openssl cms -verify -signer signcert.pem  -in test_sign.der_forged \
-inform DER  -noverify
```

where the tool `forged_attributes.py` performs the modifications to create the forged SignedData object. The code of the attack tool is found in Appendix C.

We also made a proof-of-concept implementation of the attack using the CMS verification routine of the BouncyCastle Java JCE provider. The forged message is verified correctly using the method

`org.bouncycastle.cms.SignerInformation.verify()` [2]

[2] https://www.bouncycastle.org/docs/pkixdocs1.8on/org/bouncycastle/cms/SignerInformation.html#verify-org.bouncycastle.cms.SignerInformationVerifier-

6 Impact estimation

In this section we estimate the impact of the existential forgery vulnerability. For this purpose, we first explore under which circumstances applications are not affected at all. Then we turn to the possible buildups of the forgeable messages. Finally, we attempt to qualify the likelihood for a number of conceivable malicious effects of the forgeable messages.

6.1 Applications that are not affected

Some application protocols prescribe a buildup of the CMS structure that makes the attack impossible. Specifically, this is the case when

1. the presence of the SignedAttributes in the SignerInfo is mandatory, as it is the case for instance in SCEP [Gut20], Certificate Transparency [LMS21], Firmware update according to RFC 4108 [Hou05] or the German Smart Metering CMS data format [Bun19],

2. the message is first signed and then encrypted, as then the attacker cannot learn the signature.

In these cases, no forgery can be conducted as the message will be refused during or directly after the CMS signature verification.

However, note that in case of Item 1 above, in general, the security relies on a correct implementation of the verification routine to ensure the presence of the SignedAttributes.

6.2 The structure of the forged message in the Remove-SA Attack

In order to assess the possible impact of the existential forgery, one has to consider the spectrum of the possible structures of the forged message and the degree of control that the attacker has over it.

In the Remove-SA variant, the attacker achieves a forgery in terms of a validly signed message which amounts to the DER-encoded SignedAttributes as produced by the signer during the signature generation.

Accordingly, the ASN.1 structure of the forged DER-encoded message is

```
SET {
 SEQUENCE {
       attrType OBJECT IDENTIFIER,
       attrValues SET OF AttributeValue }
 [...]
}
```

where one of the attrValues contains the message digest which is first of all variable and second potentially controlled by the attacker to a certain degree. The order of the attributes is arbitrary since for a SET the ordering is free. The inclusion of potential further attributes depends on the signer. Likely is for instance inclusion of the "useful attributes" listed in the standard [Hou09, Section 11].

The minimum number of bytes before the variable message digest, which at least in some cases the attacker might be able to influence, is 17 according to Table 2 if the messageDigest attribute is the first attribute that is encoded. The table demonstrates also the variability of the message for this case. With a different ordering of the SignedAttributes, only the tag octets of the first three fields in Table 2 are fixed, and the remaining octets depend on the (potentially proprietary) attributes chosen by the sender and their order in the SET.

Table 2: Minimal data up to and including the variable and potentially attacker controlled message digest in the forgeable message. In the left column, T, L and V refer to tag, length, and value in the ASN.1 structure, respectively. In the right column, the fixed octet values are noted as hexadecimal numbers and <L> stands for the length octets of the respective ASN.1 object.

fields	**octets**
T and L of SET	31 <L>
T and L of SEQUENCE messageDigest attribute	30 <L>
T, L, and V of the messageDigest attribute OBJECT IDENTIFIER	06 09 2A 86 48 86 F7 0D 01 09 04
T and L of the OCTET STRING	31 <L>
V of the OCTET STRING	<message digest value>

In order for the attacker to influence the value of the messageDigest attribute, he would need to be able to control a portion X of the message that is signed and further be able to predict the complete value of the resulting signed message with a sufficiently high probability in order to craft the desired message through the following process:

- in an offline preparation phase vary X randomly until

$$M' = \text{DER-encode}(\text{signedAttrs}(M(X)))$$

  has the desired form,

- let the signer sign with SignedAttributes present the message M found in this way to produce the signature S,

- and submit M' together with the valid signature S to the verifier in a SignerInfo with SignedAttributes absent.

Clearly, in the ideal setting, to achieve specific values for n octets inside messageDigest part contained in M', it is required to perform on average 2^{8n-1} hash evaluations. For a message with only 8 octets in the messageDigest chosen by the attacker, he would thus need 2^{63} hash evaluations, which is comparable to breaking DES and already a highly costly computation.

6.3 The structure of the signed message in the Add-SA Attack

In the Add-SA variant, the attacker has full control over the forged signed message, but has to achieve that the signer signs a message of the attacker's choice which is a valid DER-encoded SignedAttributes structure. In order for the produced signature to be valid for a message of his choice, it suffices if the message digest contained in the SignedAttributes is set accordingly. Since the ordering of the attributes in the structure is free and the definition of custom SignedAttributes is allowed by the specification[3], the attacker has a relatively high degree of freedom in crafting such a message, as detailed in Figure 1.

6.4 Levels of impact

In the following we estimate the impact of the two attack variants.

[3] https://datatracker.ietf.org/doc/html/rfc5652#section-2

```
31 4b 30 18 06 09 2b 80 40 80 f0 0d 01 09 03 31   // fake OID / attribute
...   30 18 06 09 2a 86 48 86 f7 0d 01 09 03 31
0b 06 09 2a 86 48 86 f7 0d 01 07 01 30 2f 06 09
2a 86 48 86 f7 0d 01 09 04 31 22 04 20 e0 be bd
22 81 99 93 42 58 14 86 6b 62 70 1e 29 19 ea 26
f1 37 04 99 c1 03 7b 53 b9 d4 9c 2c 8a
30 ...
```

Figure 1: Depiction of an exemplary DER-encoded SignedAttributes crafted by the attacker as the to-be-signed message for the Add-SA attack. The gray-colored bytes indicated fixed byte values and the light green-colored ones those that can be freely chosen by the attacker due to the ability to place a custom "fake" attribute at the beginning of the structure. The first three gray-colored bytes correspond to the ASN.1 tag values of the nested structure and thus have a fixed value. The first three dark-green-colored bytes represent the ASN.1 length fields of the nested structure with a single byte for each nesting level. Besides their value, the number of length bytes can be influenced by the attacker, and be extended from one to two or possibly more based on the size of the SignedAttributes that the attacker might arbitrarily bloat through custom attributes. At some point in the message, the mandatory attributes have to be placed, as indicated by the subsequent block of gray-colored bytes values, which represent the messageDigest attribute. The white-colored field with value 0x20 is the length of the message digest and the blue-colored range represents the value of the message digest, which the attacker can influence to some degree by iterated trial hashing.

6.4.1 Remove-SA Attack

Due to the limited flexibility of the buildup of the forged message in the Remove-SA variant of the attack, the majority of applications will not be affected in a strong sense, i.e. they will not process the forged message as a syntactically correct message.

If the forged message is fed to a parser of an application layer protocol, it may be the case that the forged message triggers security critical parser errors. The reason for the forged message passing the parsing may be due to a permissive syntax of the application layer protocol or a permissive implementation of the parser.

After passing the parsing step, the forged message is subject to interpretation. In case the message is subject to machine interpretation, according to the rather inflexible structure of a least the leading 17 bytes, it is not very likely that a targeted attack is possible in the sense that the attacker can choose to trigger a certain action. It seems rather possible that denial of service attacks could be conducted, for instance by crafting a validly signed but invalidly encoded firmware update file which brings the receiving device into an unrecoverable state. It cannot be excluded, though, that (potentially proprietary / non public) protocols exist, for which the forged messages are interpretable and lead to the triggering of harmful actions on the side of the receiving IT system.

When the message is subject to interpretation by human users, i.e., is to be read as natural language, it is rather improbable that the attacker achieves the rendering of a convincing message. Even in case the DER-encoded structure is not preventing the rendering, it is hardly feasible that the attacker will be able to craft meaningful messages through the brute-force algorithm outlined in Section 6.2.

6.4.2 Add-SA Attack

In the Add-SA variant of the attack, the impact entirely depends on whether the attacker is capable to convince the signer to sign the crafted message. If this is successful, the forged signed message has arbitrary attacker-chosen content. As shown in Section 6.3, the attacker has substantial freedom in the crafting the message. It cannot generally be precluded that he succeeds in crafting a DER-encoded SignedAttributes structure that appears as a valid file in some application context. Even certain modifications of the message by the signer can in principle be accounted for. For instance if the signer appends data of known length, this could be covered by the attacker by placing a custom attribute at the end where the ASN.1 length tag is chosen such that the structure is valid after the transformation applied by the signer.

6.5 Conceivable vulnerable systems

In this section we describe a set of conceivable vulnerable systems based on specific properties of the presumed application protocol. In general, for a protocol to be affected, it must allow optional use of the SignedAttributes structure.

6.5.1 Firmware Updates

Secure firmware updates often use signatures without encryption. If the forged message can bring a device, due to lack of robustness in the parser implementation, into an error state, this may lead to a denial of service vulnerability. The possibility of creating a targeted exploit with the Add-SA variant cannot be ruled out. However, there are three fixed values within the first few bytes of the legitimately signed message, which poses a considerable restriction on the achievable file formats.

6.5.2 Dense message space

If a protocol has a dense message space, i.e. a high probability that the forged message represents a valid command or the beginning of a valid command, then, especially if the parser is permissive with respect to trailing data, there is a risk that the message is accepted as valid. This requires a protocol where messages are signed but not encrypted.

6.5.3 Signing unstructured data

Protocols that sign unencrypted unstructured messages, e.g. tokens, might be affected in that the signature of one token might result in the corresponding forged message being another valid token.

6.5.4 External signatures over unstructured secret data

The class of systems that is possibly strongest affected by the Remove-SA attack would be one that uses external signatures[4], i.e. CMS signatures with absent content (that may be transmitted encrypted separately) over unstructured data, e.g. a token of variable length. In that case the attacker could derive from such a signature with SignedAttributes present a signed data object for a secret known to him.

7 Mitigation

In this section we discuss necessary and possible mitigation measures.

[4] https://datatracker.ietf.org/doc/html/rfc5652#section-5.2

7.1 Mitigation in the CMS standard

To fundamentally remove the vulnerability, the CMS standards need to be updated. The signature verification needs to be unambiguous with respect to what is the signed message. One solution would for instance be to mandate the unconditional presence of the SignedAttributes.

7.2 Mitigation in client protocols

Protocols that use CMS achieve security if they strictly prohibit the use the content type "id-data" in the SignerInfo structure. The reason for this is that, as explained above, only when the content type is "id-data", the presence of the SignedAttributes is optional. For all other content types their presence is mandatory. Nevertheless, secure verification is achieved by an implementation only if it enforces the presence of the SignedAttributes during signature generation and verification.

7.3 Attack detection as a mitigation in CMS implementations

Software libraries that implement the CMS signature generation and verification may consider to apply an attack detection mechanism to the signed message. Namely, they would check if the signed message is a valid DER-encoded SignedAttributes structure and in this case

- refuse the signature generation if the SignedAttributes are not set in the SignerInfo, thus preventing the Add-SA attack,
- let the signature verification fail if the SignerInfo does not contain the SignedAttributes, thus preventing the Remove-SA attack.

This is clearly not standard conforming, as in the case an application protocol would deliberately use such a structure as the signed message[5], it would invalidly fail. However, it is very unlikely that any application uses a message exactly matching encoded SignedAttributes.

The algorithmic description of a possible implementation of the detection countermeasure is given in Appendix B.

7.4 Mitigation in application protocols

Application protocols that not already do so should be updated to always require the presence of the SignedAttributes during the signature generation and verification.

7.5 Mitigation in application implementations

Application implementations that cannot enforce the presence of the SignedAttributes in the SignerInfo during verification should be ensured to behave robustly when receiving as input a message of the structure of SignedAttributes and discard this message early during the processing. Furthermore, it should be considered to use the attack detection mechanism described in Section 7.3.

[5] Even though the buildup of the general ASN.1 structure of the SignedAttributes may coincide with other structures, the presence of the combination of the contentType and messageDigest attributes, identifiable through their respective object identifiers, that are both mandatory in CMS SignedAttributes, should never occur in other contexts and thus should allow for identification of the SignedAttributes with rather high confidence.

8 Conclusion

In this work we describe two variants of an existential signature forgery attack against the widely deployed CMS protocol. Signature forgeries are generally a severe type of vulnerability as they, among other possible effects, might allow to bypass authentication. In the present case, the effect seems to be limited in the majority of the presumable applications, since in the Remove-SA variant, the structure of forged message is rather inflexible and in the Add-SA variant, the beginning of intentionally signed message is subject to considerable restrictions. However, due to the presumably vast number of applications built on the basis of CMS or its predecessor PKCS#7, the existence of vulnerable systems cannot be precluded.

We describe several approaches to mitigations. Especially the attack detection mechanism during the signature verification described in Section 7.3 seems to be a quick and almost unconditionally applicable solution until a revision of the standards has taken place.

References

[Bun19] Bundesamt für Sicherheit in der Informationstechnik. Technische Richtlinie BSI TR-03109-1 – Anlage I: CMS-Datenformat für die Inhaltsdatenverschlüsselung und -signatur, 2019. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03109/TR-03109-1_Errata-CMS.pdf?__blob=publicationFile&v=9.

[Gut20] P. Gutmann. RFC 8894 – Simple Certificate Enrolment Protocol, 2020. https://datatracker.ietf.org/doc/html/rfc8894.

[Hou05] Russ Housley. RFC 4808 – Using Cryptographic Message Syntax (CMS) to Protect Firmware Packages, 2005. https://datatracker.ietf.org/doc/html/rfc4108.

[Hou09] Russ Housley. RFC 5652: Cryptographic Message Syntax (CMS), 2009. https://tools.ietf.org/html/rfc5652.

[Kal98] Burt Kaliski. RFC 2315 – PKCS #7: Cryptographic Message Syntax Version 1.5, 1998. https://datatracker.ietf.org/doc/html/rfc2315.

[LMS21] B. Laurie, E. Messeri, and R. Stradling. RFC 9162 – Certificate Transparency Version 2.0, 2021. https://www.rfc-editor.org/rfc/rfc9162.html.

[STR19] J. Schaad, S. Turner, and B. Ramsdell. RFC 8551 – Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification , 2019. https://tools.ietf.org/html/rfc8551.

Appendix

A SignedAttributes binary data

Here we present a sample DER-encoded SignedAttributes object for reference in the form as they are signed. Note that they are encoded without the `[0] IMPLICIT` tag that they carry when embedded into a SignerInfo structure.

The sample SignedAttributes value in hexadecimal notation:

```
31 4b 30 18 06 09 2a 86  48 86 f7 0d 01 09 03 31
0b 06 09 2a 86 48 86 f7  0d 01 07 01 30 2f 06 09
2a 86 48 86 f7 0d 01 09  04 31 22 04 20 e0 be bd
22 81 99 93 42 58 14 86  6b 62 70 1e 29 19 ea 26
f1 37 04 99 c1 03 7b 53  b9 d4 9c 2c 8a
```

And the result of an ASN.1 parse thereof:

```
   <31 4B 30 18 06 09 2A 86 48 86 F7 0D 01 09 03 31 0B 06 09 2A 86 48 86 F7>
 0  75: SET {
   <30 18 06 09 2A 86 48 86 F7 0D 01 09 03 31 0B 06 09 2A 86 48 86 F7 0D 01>
 2  24:   SEQUENCE {
   <06 09 2A 86 48 86 F7 0D 01 09 03>
 4   9:     OBJECT IDENTIFIER contentType (1 2 840 113549 1 9 3)
   <31 0B 06 09 2A 86 48 86 F7 0D 01 07 01>
15  11:     SET {
   <06 09 2A 86 48 86 F7 0D 01 07 01>
17   9:       OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
      :       }
      :     }
   <30 2F 06 09 2A 86 48 86 F7 0D 01 09 04 31 22 04 20 E0 BE BD 22 81 99 93>
28  47:   SEQUENCE {
   <06 09 2A 86 48 86 F7 0D 01 09 04>
30   9:     OBJECT IDENTIFIER messageDigest (1 2 840 113549 1 9 4)
   <31 22 04 20 E0 BE BD 22 81 99 93 42 58 14 86 6B 62 70 1E 29 19 EA 26 F1>
41  34:     SET {
   <04 20 E0 BE BD 22 81 99 93 42 58 14 86 6B 62 70 1E 29 19 EA 26 F1 37 04>
43  32:       OCTET STRING
      :         E0 BE BD 22 81 99 93 42 58 14 86 6B 62 70 1E 29
      :         19 EA 26 F1 37 04 99 C1 03 7B 53 B9 D4 9C 2C 8A
      :       }
      :     }
      :   }
```

B Attack detection countermeasure

In this section we provide an algorithmic description of a possible implementation of the attack detection countermeasure described in Section 7.3. The purpose is to detect a potential attack during the signature generation and verification. Algorithm 3 determines whether the input data fulfills the minimal condition to be a valid SignedAttributes structure. It is used in the secure verification routine, given in Algorithm 4 and in the signature generation routine, given in Algorithm 5.

C Attack tool in Python for the Remove-SA Attack

In Listing C.1 we give an attack tool that performs the Remove-SA attack variant.

Algorithm 3 Routine for determining that a given byte array is a valid DER-encoded SignedAttributes structure. If one of the instructions indicated as "ensure ..." is not fulfilled, the algorithm breaks with result *false*

Algorithm IS-ATTACK-MESSAGE(message M)
    contentTypeFound = *false*
    messageDigestFound = *false*
    ensure M is decodable as a SET
    set = M
    for all elements *seq* in *set* **do**
        ensure *seq* is decodable as a SEQUENCE
        ensure *seq* contains two elements
        ensure *first element in seq* is of type OBJECT IDENTIFIER
        attrOID = *first element in seq*
        ensure *second element in seq* is of type SET
        attrSet = *second element in seq*
        if *attrOID* has value `2A 86 48 86 F7 0D 01 09 03` **then**
            ensure *attrSet* contains a single element of type OBJECT IDENTIFIER
            contentTypeFound = *true*
        end if
        if *attrOID* has value `2A 86 48 86 F7 0D 01 09 04` **then**
            ensure *attrSet* contains a single element of type OCTET STRING
            messageDigestFound = *true*
        end if
    end for
    if contentTypeFound == *true* AND messageDigestFound == *true* **then**
        return *true // attack messsage detected*
    end if
    return *false*
end Algorithm

Algorithm 4 Attack detection during signature verification.

Algorithm SECURE-VERIFY(message M, signature S, public key K_p, *signedAttrsPresent* (boolean))
    if *signedAttrsPresent* **then**
        perform the verification according to Algorithm 2
    end if
    if IS-ATTACK-MESSAGE(M) **then**
        return *false*
    end if
    D = HASH(M)
    return verify(K_p, D, S)
end Algorithm

Algorithm 5 Attack detection during signature generation.

Algorithm SECURE-SIGN(message M, secret key K_s, *signedAttrsPresent* (boolean))
  if *signedAttrsPresent* **then**
    perform the signature generation according to Algorithm 1
  end if
  if IS-ATTACK-MESSAGE(M) **then**
    abort with error "invalid message"
  end if
  $D = \text{HASH}(M)$
  return sign(K_p, D, S)
end Algorithm

```
#!/usr/bin/python3

import sys, os
from asn1crypto import cms, util, core
import binascii


def main():
    if len(sys.argv) != 2:
        print("missing input file")
        print("call with --help for more information")
        sys.exit(1)
    if sys.argv[1] == "--help":
        print("how to run attack:")
        print("openssl cms -sign -nodetach -signer signcert.pem -inkey signcert_pri.pem -binary -in data.file -outform
      DER -out test_sign.der")
        print("./forged_attributes.py test_sign.der")
        print("openssl cms -verify -signer signcert.pem -in test_sign.der -inform DER -noverify")
        print("openssl cms -verify -signer signcert.pem -in test_sign.der_forged -inform DER -noverify")
        exit(0)
    with open( sys.argv[1], 'rb') as f:
        info = cms.ContentInfo.load(f.read())
        signed_data : cms.SignedData = info['content']
        signer_infos : cms.SignerInfos = signed_data['signer_infos']
        for signer_info in signer_infos:
            new_content : cms.CMSAttributes = signer_info['signed_attrs']
            signer_info['signed_attrs'] = None
        new_content_bytes = bytearray(new_content.dump())
        new_content_bytes[0] = 0x31
        print("new_content_bytes = " + str(binascii.hexlify(bytearray(new_content_bytes)).decode('ascii')))

        signed_data['encap_content_info']['content']= core.OctetString(bytes(new_content_bytes))
        with open(sys.argv[1] + '_forged','wb+') as fout:
            fout.write(info.dump())

if __name__ == "__main__":
    main()
```

Listing C.1: Attack tool