ForgedAttributes: An Existential Forgery Vulnerability of CMS and PKCS#7 Signatures

v1.5, November 22, 2023

Falko Strenzke

MTG AG
`falko.strenzke@mtg.de`

Abstract. This work describes an existential signature forgery vulnerability of the current CMS and PKCS#7 signature standards. 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. If an attacker takes a CMS or PKCS#7 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. Due to the limited flexibility of the forged message and the limited control the attacker has over it, the fraction of vulnerable systems must be assumed to be small but due to the wide deployment of the affected protocols, such instances cannot be excluded. We propose a countermeasure based on attack-detection that prevents the attack reliably.

Table of Contents

1 Introduction

The CMS [1] and PKCS#7 [2] cryptographic data formats, both being very similar to each other, are among the most widely used. Among other protocols, they are employed for instance in S/MIME [3] and protocols for certificate management [4,5]. 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 so-called SignedAttributes field in 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. The message digest is contained in these attributes and is thereby verified. However, the attacker can simply remove them to cause the verifier to feed the message directly to the message digest. Accordingly, encoded SignedAttributes themselves are a validly signed message if the SignedAttributes field is absent from the structure.

2 CMS Signatures

CMS signatures are defined in [1]. 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 optional presence of the SignedAttributes. If they are included in the SignerInfo structure is, according to the CMS standard, at discretion of the signer. If they 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 as its value of the signed message.

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 $\text{sign}(K_s, D)$
9: end Algorithm
```

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

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.

```
1: Algorithm CMS-VERIFY(message $M$, public key $K_p$, signedAttr$_M^{\text{DER}}$, signature $S$)
2:     $D = \text{HASH}(M)$
3:     if signedAttr$_M^{\text{DER}} = \emptyset$ then
4:         $E = D$
5:     else
6:         $F$ = message digest for $M$ contained in signedAttr$_M^{\text{DER}}$
7:         if $F \neq D$ then
8:             return FAIL
9:         end if
10:        $E = \text{HASH}(\text{signedAttr}_M^{\text{DER}})$
11:    end if
12:    return verify$(K_p, E, S)$
13: end Algorithm
```

3 PKCS#7 signatures

PKCS#7 signatures are defined in [2]. 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.

4 Existential signature forgery attack

An existential signatures attack against CMS or PKCS#7 signatures can be conducted as follows. The signer signs a 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

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

that was never signed by the signer.

[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.

5 Proof-of-concept implementation

We created an attack tool that executes the 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]

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 or PKCS#7 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 [4], Certificate Transparency [5], Firmware update according to RFC 4108 [6] or the German Smart Metering CMS data format [7],

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

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 or PKCS#7 signature verification.

However, note that in case of Item 1 above the security relies on a correct implementation of the verification routine to ensure the presence of the SignedAttributes which is not the case without the existential forgery vulnerability if there are only trusted signers who produce correct SignerInfos for the respective protocol.

6.2 The structure of the forged message

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

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 the "useful attributes" listed in the standard [1, Section 11].

The minimum number of bytes before the variable message digest is 17 according to Table 1 if the messageDigest attribute is the first attribute that is encoded. The table demonstrates also the variability of the message for this case, namely the length octet(s) <L> in the right column are variable according to the content length.

However, with a different ordering of the SignedAttributes, only the tag octets of the first three fields in Table 1 are fixed, and the remaining octets depend on the (potentially proprietary) attributes chosen by the sender and their order in the SET.

In order for the attacker to control 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

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

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>

Table 1: 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.

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', he needs 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 Levels of impact

Due to the limited flexibility of the buildup of the forged message, 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 Conceivable vulnerable systems

In this section we describe a set of conceivable vulnerable systems based on specific properties of the presumed application protocol. Thereby, we have of course the precondition that the affected protocol must allow for the absence of the SignedAttributes during verification.

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 can be excluded with greatest certainty in this case due to the lack of control the attacker has over the forged message.

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.

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.

External signatures over unstructured secret data The probably strongest affected class of systems would be one that uses external signatures[3], i.e. CMS or PKCS#7 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 create a signed data object for a known secret.

7 Mitigation

In this section we discuss necessary and possible mitigation measures.

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

7.1 Mitigation in the CMS and PKCS#7 standards

In order to remove the vulnerability, the 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 always use the SignedAttributes as the input to the message digest.

7.2 Attack detection as a mitigation in CMS and PKCS#7 implementations

Software libraries that implement the CMS and PKCS#7 signature verification may consider to apply an attack detection mechanism to the signed message in the case of the SignerInfo not carrying the SignedAttributes. Namely, they would check if the signed message is a valid DER-encoded SignedAttributes structure and in this case let the signature verification fail. This is clearly not standard conforming, as in the case an application protocol would deliberately use such a structure as the signed message[4], it would invalidly fail. However, on the one hand, it is very unlikely that any application uses a message exactly matching encoded SignedAttributes. Furthermore, the CMS and PKCS#7 signing routines of the libraries could also test for the message being a valid DER-encoded SignedAttributes structure and thus indicate to the application early that this type of message is refused.

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

7.3 Mitigation in application protocols

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

7.4 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 7.2.

[4]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 and PKCS#7 SignedAttributes, should not often occur in other contexts and thus should allow for identification of the SignedAttributes with rather high confidence.

7.5 Signing with SignedAttributes absent

Clearly, if the signer signs his messages with SignedAttributes absent, he prevents the attack. However, using this as an approach to mitigate the attack should be considered with care, as it is not compatible with the more future-proof solution to enforce the presence of the SignedAttributes during verification.

8 Conclusion

In this work we show an existential signature forgery attack against two widely deployed security protocols. Signature forgeries are generally a severe type of vulnerability as they, among other possible effects, allow to bypass authentication. In the present case, the effect seems to be limited in the majority of the presumable applications, since the buildup of the forged message is rather inflexible. However due to the presumably vast number of applications build on these security protocols, 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 7.2 seems to be a quick and almost unconditionally applicable solution until a revision of the standards has taken place.

References

1. R. Housley: RFC 5652: Cryptographic Message Syntax (CMS) (2009) `https://tools.ietf.org/html/rfc5652`.
2. B. Kaliski: RFC 2315 – PKCS #7: Cryptographic Message Syntax Version 1.5 (1998) `https://datatracker.ietf.org/doc/html/rfc2315`.
3. J. Schaad, S. Turner, B. Ramsdell : RFC 8551 – Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification (2019) `https://tools.ietf.org/html/rfc8551`.
4. P. Gutmann: RFC 8894 – Simple Certificate Enrolment Protocol (2020) `https://datatracker.ietf.org/doc/html/rfc8894`.
5. B. Laurie, E. Messeri, R. Stradling: RFC 9162 – Certificate Transparency Version 2.0 (2021) `https://www.rfc-editor.org/rfc/rfc9162.html`.
6. R. Housley: RFC 4808 – Using Cryptographic Message Syntax (CMS) to Protect Firmware Packages (2005) `https://datatracker.ietf.org/doc/html/rfc4108`.
7. 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`.

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 7.2. The purpose is to detect a potential attack during the signature verification and, to facilitate interoperability, to prevent the generation of signed messages that would be refused by a recipient who implements the attack detection countermeasure.

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, to protect against the attack and in the signature generation routine, given in Algorithm 5, to prevent the signer from signing data that would not be veriable by a verifier that uses the attack detection.

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*

```
 1: Algorithm IS-ATTACK-MESSAGE(message $M$)
 2:     contentTypeFound = false
 3:     messageDigestFound = false
 4:     ensure $M$ is decodable as a SET
 5:     set = $M$
 6:     for all elements seq in set do
 7:         ensure seq is decodable as SEQUENCE
 8:         ensure seq contains two elements
 9:         ensure first element in seq is of type OBJECT IDENTIFIER
10:         attrOID = first element in seq
11:         ensure second element in seq is of type SET
12:         attrSet = second element in seq
13:         if attrOID has value 2A 86 48 86 F7 0D 01 09 03 then
14:             ensure attrSet contains a single element of type OBJECT IDENTIFIER
15:             contentTypeFound = true
16:         end if
17:         if attrOID has value 2A 86 48 86 F7 0D 01 09 04 then
18:             ensure attrSet contains a single element of type OCTET STRING
19:             messageDigestFound = true
20:         end if
21:     end for
22:     if contentTypeFound == true AND messageDigestFound == true then
23:         return true // attack messsage detected
24:     end if
25:     return false
26: end Algorithm
```

C Attack tool in Python

In Listing C.1 we give the attack tool.

Algorithm 4 Attack detection during signature verification. This algorithm prevents the signature forgery attack.

```
1: Algorithm SECURE-SIGNATURE-VERIFY(message $M$, signature $S$, public key $K_p$,
   signedAttrsPresent (boolean))
2:     if signedAttrsPresent then
3:         perform the verification according to Algorithm 2
4:     end if
5:     if IS-ATTACK-MESSAGE(M) then
6:         return false
7:     end if
8:     $D = \text{HASH}(M)$
9:     return verify$(K_p, D, S)$
10: end Algorithm
```

Algorithm 5 Prevention of signing invalid messages during signature generation. This algorithm prevents the generation of signed messages that will be refused by a verifier who implements Algorithm 4.

```
1: Algorithm ROBUST-SIGN(message $M$, secret key $K_s$, signedAttrsPresent
   (boolean))
2:     if signedAttrsPresent then
3:         perform the signature generation according to Algorithm 1
4:     end if
5:     if IS-ATTACK-MESSAGE(M) then
6:         abort with error "invalid message"
7:     end if
8:     $D = \text{HASH}(M)$
9:     return sign$(K_p, D, S)$
10: 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