Pretty Good Piggy-backing

Parsing vulnerabilities in PGP Desktop

Eric R. Verheul

Digital Security group, Radboud University Nijmegen

Security & Technology group, PwC

Eric.Verheul@[cs.ru.nl, nl.pwc.com]

Summary

In this paper we demonstrate 'piggy-back' attacks on PGP Desktop 10 and its predecessors which can be exploited in targeted cybercrime attacks, i.e. targeting specific influential persons within an organization. We show that an attacker can add PGP messages (e.g. malicious files) into existing PGP messages signed by trusted sources in such a way that PGP Desktop still indicates that the result is signed while the decryption includes the additional messages. This could be exploited by an active attacker adding malicious content (e.g., executables or incriminating content such as pictures) into a PGP message: the result is still indicated authentic by PGP Desktop but the decrypted content will not only contain the legitimate message but also the malicious content.

Another type of exploitation can be based on manipulating the filenames contained in PGP messages indicating where PGP Desktop should store the results in. As these filenames are not signed even when the file itself is, an attacker can rename the originally singed file (e.g., a contract) to something innocent looking (e.g., a Windows temporary file) and add a changed version of the file to the PGP message. The recipient will see the signature of the legitimate file but will not get the file itself; instead it will get the changed file. That is, the recipient will trust the authenticity of the changed file based on the signature on the original file.

We relate these attacks to the OpenPGP specifications (RFC 4880) and the lack of parsing rules therein on multi packet messages. We also discuss how OpenPGP messages can be used as a hidden channel, i.e. to contain information hidden from recipients and senders. This typically applies to all applications processing OpenPGP messages, e.g. also the GNU Privacy Guard (GPG).

Despite the long established PGP open source policy these vulnerabilities were apparently not publicly noticed, which might lower the trust in the 'security-by-public-inspection' philosophy of open source software. We finally suggest some security improvements to the OpenPGP specifications.

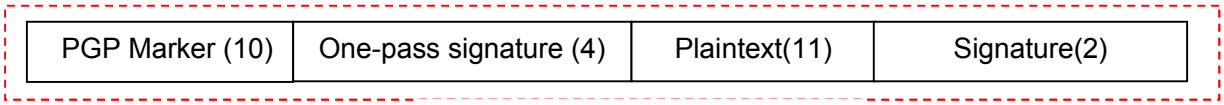
1. Introduction

The Pretty Good Privacy (PGP) program was the first freely and publicly available implementations of public-key cryptography written in 1991 by Phil Zimmerman. Its source code was open so that users could assure themselves it did not contain bugs and/or backdoors. In 2002 the PGP Corporation was founded that acquired the source code of PGP. The PGP Corporation developed a series of security products based on the PGP source code including further improvements on the PGP product itself as well as new products. With the acquisition PGP gradually became less free software, but all PGP Desktop software still provide free basic encryption and decryption functionality for non-commercial use after the trial period has ended. Also the practice of publication of source code was still adhered to by PGP Corporation . Although no longer supported by PGP Corporation, version 8 of PGP Desktop is still commonly used as (unauthorized) copies can be

easily found on the internet. Versions 9 and 10 are still supported by PGP Corporation. In April 2010, PGP Corporation was acquired by Symantec.

Several other public-key cryptography based programs find their basis in PGP, the most prominent being the GNU Privacy Guard or GPG. PGP was also the driving force between the open (de facto) standard RFC 4880 in which the OpenPGP messages are specified. These specifications are adhered to by all 'PGP' implementations most notably GPG and the PGP products developed by PGP Corporation.

Any OpenPGP message consists of a number of records that are traditionally called packets. A packet is a chunk of data that has a tag specifying its meaning. Packets themselves consist of a header and a body; the header specifies its tag and the length of the body and the body contains the actual data. As an illustration the PGP Marker packet (used within PGP products) is a series of 5 hexadecimal bytes A3 03 50 47 50. Here 'A3' describes the version of the packet and its tag number (10) and '03' describes the length of the body (which is 3). The numbers '50 47 50' correspond to the ASCII representation of PGP. There are about 60 different types of packets. Below we have illustrated the form of a signed PGP message. Here the One-pass signature packet tells the processing software amongst other things which hash function to use when passing through the plaintext packet (without the One-pass signature packet the software would need to pass two times through the OpenPGP message).



PGP signed message

As an OpenPGP message typically consists of several packets software that processes such messages should parse them. In this paper we demonstrate that parsing is security relevant: the sloppy parsing by PGP Desktop (versions 8 - 10) introduces serious security vulnerabilities. This demonstration leads to the conclusion that there should be 'proper' parsing guidelines in the OpenPGP specifications which are currently not provided.

This paper is outlined as follows:

- In Section 2 we demonstrate the security vulnerabilities that currently exist in OpenPGP desktop (version 10) and its predecessors related to record parsing.
- In Section 3 we discuss some issues related to (rigorous) OpenPGP packet parsing.
- In the Appendix we report in detail on the experiments we have done with OpenPGP desktop (version 10) that form the basis for Section 2. This appendix ensures that our experiments can be reproduced.

The experiments in the Appendix are based on PGP Desktop (version 10) in the Windows environment and as far as we can verify the vulnerabilities also exist in its predecessors versions 8 and 9 as well. We have not tested other operating systems than Windows. We finally note that we have not tested other PGP Corporation products processing OpenPGP messages such as "PGP Command Line" and "PGP Netshare".

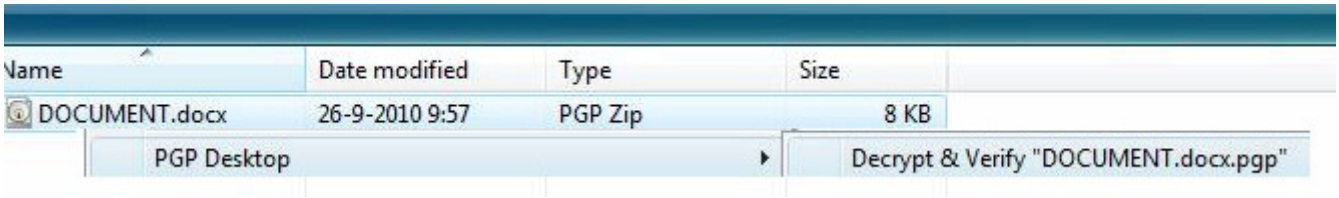
Acknowledgements

Flavio Garcia is thanked for the stimulating discussions on security issues in the OpenPGP specifications. GOVCERT.NL is thanked for their comments and their independent verification of the experiments performed in the appendix using PGP Desktop 9.8.3.

2. OpenPGP parsing vulnerabilities in PGP Desktop version 10

2.1 Background

PGP Desktop provides different methods for users to use its cryptographic functions. The method we focus on in this paper is the one provided to the user through the Windows file explorer. Here the user clicks on a file, typically with the right mouse button as indicated in the picture below (we assume the user is a right handed). The user then gets the option to "Decrypt & Verify" that file. After selecting this option, files are decrypted, signatures are verified and the results are shown to the user. "Decrypt & Verify" was already part of earlier PGP Desktop versions (such as version 8) and we believe this is a popular way of using PGP Desktop.



We consider the context of three persons Sender, Receiver and Eavesdropper. The Sender and Receiver both have created PGP keys and have securely shared the appropriate public parts among them. The Sender is sending PGP messages (encrypted, signed or both) to the Receiver but they are intercepted and stopped by Eavesdropper who has the opportunity to manipulate these messages and send them through to Receiver after this. The general idea is that Receiver will not suspect anything as long as "Decrypt & Verify" does not complain.

In the descriptions below and in the experiments we only 'piggy-back' one file but one can actually 'piggy-back' any number of files. We also note that all OpenPGP messages we 'piggy-back' do not contain the PGP Marker packet (see above) which are routinely added by PGP Desktop as the first packet. This marker is not mandatory but PGP Desktop refuses to process a PGP message where the market packet occurs at a later position than the first position. GPG does not add such markers and this was one of the reasons of using GPG for constructing ciphertexts.

2.1 Piggy-backing additional content

In the first scenario Sender is sending an OpenPGP ciphertext C to Receiver, consisting of an encrypted file M (e.g. a Word document) under the public key of the Receiver and signed with his (i.e. Sender) private key. This means that the encrypted file roughly consists of an OpenPGP packet holding the message symmetrically encrypted under a random session key, a packet holding the session key encrypted under a public key and a packet holding a signature over the message.

Eavesdropper now encrypts some malicious file B with the public key of Receiver and just places it after the end of the ciphertext C, leading to C'. Note that typically such keys are published so it would not be difficult for Eavesdropper to obtain these keys. File B could for instance be a malicious executable, a Windows shortcut or incriminating content.

Next Eavesdropper sends C' to Receiver that uses "Decrypt & Verify" on it. Now note that in C' there is a signed and encrypted message and an (only) encrypted message M added by Eavesdropper. The proper way for the "Decrypt & Verify" option would be to communicate this to the user, i.e. 'there is one signed and encrypted message in C' and one message that is only encrypted'. Of course "Decrypt & Verify" could also just refuse to process ciphertext C' (as GPG typically does). But what "Decrypt & Verify" actually does is quite bizarre: it decrypts both files (leading to two files M and B), stores them and states that the whole (!) message is signed. See Experiments #1 and #2 for details. In the first experiment B is a malicious file and in the second experiment B is a message for the so-called PGP secure viewer.

If C is only signed by Sender (and not encrypted for Receiver) then Eavesdropper could also add a file encrypted for Receiver to C as before. But we also show in Experiment #3 that he could add a plaintext file B after C so that Receiver would not need to use his private key to decrypt C'. "Decrypt & Verify" behaves as before and it states that the whole message is signed and stores both M and B on the user's computer.

2.2 Piggy-backing and renaming content

The OpenPGP specifications support inclusion of the filename to be used for the end result (e.g., a decrypted file). This has some potential vulnerabilities (e.g., when the included file refers to an arbitrary location in the file system, e.g., c:\Windows\System32\cmd.exe). But these risks seem to be properly addressed in PGP Desktop: one seems not to be able to automatically store a file at a location different from that of the ciphertext file. A more fundamental vulnerability in the OpenPGP specifications is that the filename (or the length of the filename) provided by the sender is not signed, even when the file itself is signed. Moreover, PGP Desktop introduces an extra vulnerability by automatically using this filename when it is provided in the ciphertext. Together with the described piggy-back issues this introduces some interesting attack possibilities.

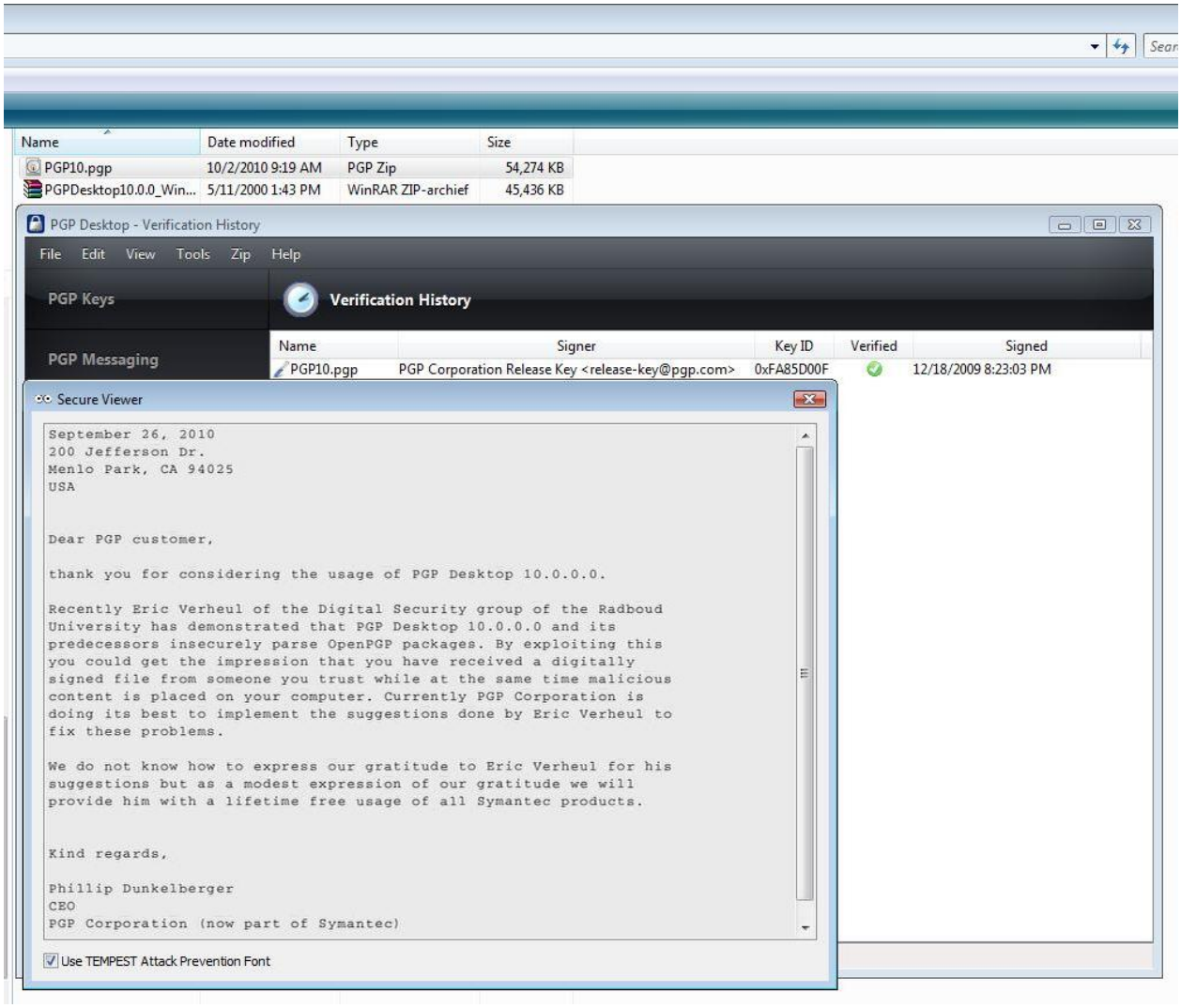
In the Experiment #5 we start with a signed file (e.g. containing a contract), which we cloak by changing its name to something innocent looking, e.g. a Windows temporary file. Then we add as plaintext a file with a changed version of that file (e.g. a changed contract) with the original name. The recipient will see the signature of the legitimate file but will not get the file itself; instead it will get the changed file. That is, the recipient will trust the authenticity of the changed file based on the signature on the original file. In this experiment we have replaced an original Word document with another Word document. One can also use a Windows 'shortcut' file (extension .lnk) and giving it the same icon as a Wordpad document. This might lure the recipient into double clicking the 'shortcut' file which contains malicious content, e.g. a command line script that will be executed when the file is double clicked. The script could actually cover up its tracks and start the legitimate Wordpad file.

2.3 Piggy-backing with detached signatures

In this context Sender is sending the original file M and a separate (detached) signature S. This is quite commonly used in software distribution, i.e. when M is a software installation file. Now the strategy of Eavesdropper is to combine the file M and the detached signature S into one validly signed OpenPGP message and then to use the techniques we have described in the previous section. In Experiment #4 we demonstrate that this is also possible.

In this experiment we first create – as in Experiment #3 – the plaintext packet (tag 11) using GPG and GPGSPLIT. But one cannot just add the detached signature behind the plaintexts packet as the OpenPGP format expects a so-called One-pass signature packet (tag 4) before the plaintext and the signature. The One-pass signature packet tells the processing software amongst other things which hash function to use when passing through the plaintext packet (without the One-pass signature packet the software would need to pass two times through the OpenPGP message). In Experiment #4 we create this One-pass signature packet with GPG and GPGSPLIT and some manual editing. We set some special parameters in the usage of GPG to make up for the different default behavior between PGP and GPG.

The PGP Desktop distribution itself also uses a detached signature and we have used the technique from Experiment #3 to create a message for the secure viewer that allegedly was signed by the PGP Corporation. This is depicted in the screen capture below.



2.4 Using signatures from the 'Web of Trust'

In the context of PGP the secure binding of identities with public keys is not based on digital certificates as specified in the X.509 standard. Instead PGP public keys are typically signed by other PGP users that vouch for the public keys of others by signing their public keys. This forms the so-called PGP Web of Trust and these signed keys can then be placed on PGP key servers. The exploitations we have done so far are all based on existing digital signatures (either detached or not) on normal content. That is, to perform them an attacker needs existing digital signatures on normal content from the senders he likes to spoof. An interesting harvesting method would be to try to use the signatures that are in the PGP Web of Trust. However an analysis of the OpenPGP specifications indicate that this is not possible. In these specifications it is stipulated that the 'signature type' is part of the data signed and the signature type of normal content (typically type 0x00) is different from the signature type used for public keys (typically 0x11).

2.5 Hidden channels in the OpenPGP specifications

Additionally we note that even content inside OpenPGP messages that is not stored to the file system might introduce security vulnerabilities. Indeed, several packets in the OpenPGP specifications (most notably packets 61 and 63) are commonly ignored by OpenPGP applications. In addition, 'white spaces' between armored text lines are ignored by applications. This means that these packets (and armored texts such as PGP public keys) could provide for a hidden channel from a high secure environment without the recipient (or sender) noticing. OpenPGP processing software might consider warning the user on the occurrence of such anomalies.

3. Conclusion

In this paper we have demonstrated that applications should be careful in handling OpenPGP messages, specifically those that lead to more than one output message. If an application wants to provide such multi-message support at the very least it should specify the status of each of the messages to the user in detail, e.g., to distinguish the signed from the non-signed messages. Actually, in situations where security is critical it might be sensible to only store the results to the file system after explicit approval of the user to avoid 'one-click attacks' where only selecting the file, e.g. to delete it, will already run malicious code. The user should also be explicitly asked if the filename suggested in the ciphertext should be used for storing the results to the file system. In this respect we think that the OpenPGP specifications (RFC 4880) should enforce that if a file is signed any suggested filename should be part of the signature too.

We additionally noted that even content inside OpenPGP messages that is not stored to the file system might introduce security vulnerabilities. Indeed, several packets in the OpenPGP specifications provide for a hidden channel. OpenPGP processing software might consider warning the user on the occurrence of such anomalies.

In summary, we suggest that the OpenPGP specifications (RFC 4880) are supplemented with security policies on multi-packet parsing. In line with this we suggest a 'hardened PGP' policy is introduced that only allows the most basic OpenPGP packets to be part of a PGP message and stipulates secure processing of the allowed packets. To illustrate, OpenPGP processing software following that policy should allow the user inspection of the contents and should only store results to the file system after explicit approval by the user, it should also warn the user on the occurrence of possible hidden information.

Appendix: PGP Desktop piggy-backing experiments
Below we describe the experiments we have conducted in detail. Instructions start with a reference of type [X] where X can either be S for Sender, E for Eavesdropper and R for Receiver. To avoid interference in PGP installations we have used separate computers for the Sender, Eavesdropper and the Receiver (see below). Moreover, we have used a separate file server to mimic the 'internet'.

General preparatory work

- [R,S]: Install 32 bit PGP Desktop 10.0.0 on two desktops (D_R for Recipient and D_S for Sender), both based on Windows Vista.
- [E]: Install GPG 2.0 on a laptop L_E (representing active Eavesdropper of messages from D_S to D_R), based on Windows XP professional. This includes GPGSPLIT.
- [S]: Generate a public/private key pair for Sender on desktop D_S (selected a passphrase ("sender") to avoid automatic decryption); export public key and make available to D_R and L_E
- [R]: Generate a public/private key pair for Recipient on desktop D_R (selected a passphrase ("recipient") to avoid automatic decryption); export public key and make available to D_S and L_E. Actually the public key of Sender is not required for L_E.

Preparatory work for Experiments #1 and #2

- [S]: Create a Word file (TextStoR.docx) on D_S pertaining a message from Sender to Recipient.
- [S]: PGP encrypt "TextStoR.docx" with Public key of D_R and signed with D_S's private key, leading to file "TextStoR.docx.pgp".
- [S]: 'Send' file "TextStoR.docx.pgp" from D_S tot D_R but this is actively intercepted by L_E. That is, this file is manipulated as indicated below and sent through to D_R afterwards.

Experiment #1: basic piggy-backing malicious content

[E]: GPG encrypt malicious executable mal.exe with public key of D_R, leading to file mal.exe.gpg.

```
C:\Program Files\GNU\GnuPG>gpg -e -r recipient mal.exe
gpg: 83E49FEB: There is no assurance this key belongs to the named user

pub  2048R/83E49FEB 2010-09-25 Recipient <recipient@recipient.nl>
 Primary key fingerprint: B8F8 1D44 36F9 16BD 495C  D44C F582 885B 7DA3 FFA1
      Subkey fingerprint: E938 4E5C DF90 7E83 6F13  31A3 F980 86F4 83E4 9FEB

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N)
```

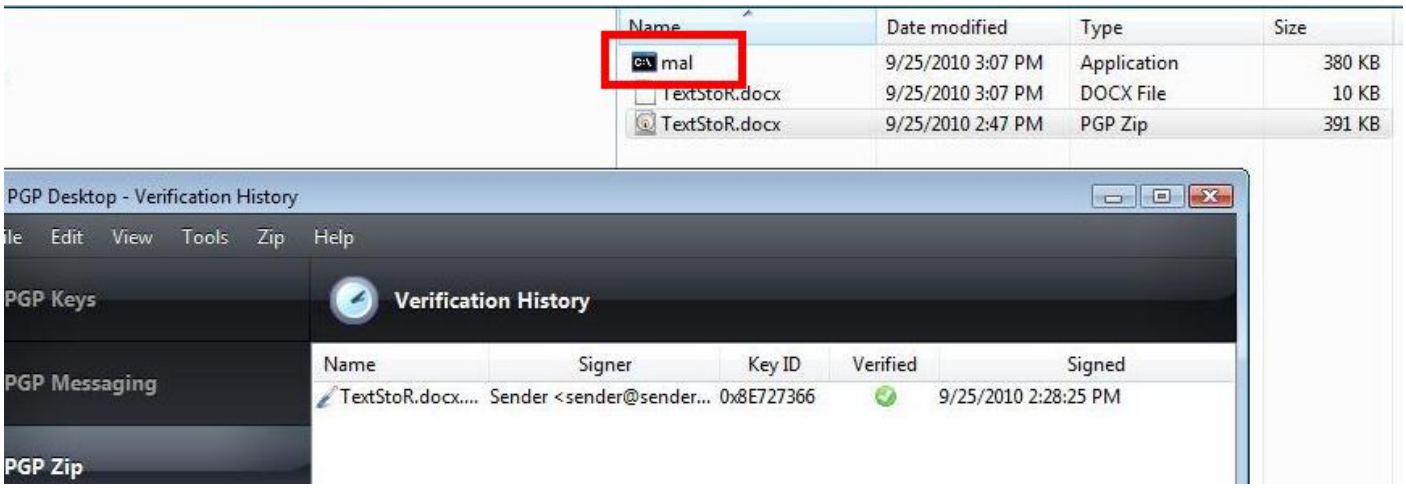
[E]: Concatenate mal.exe.gpg to intercepted file "TextStoR.docx.pgp", leading to a (large) file named "TextStoR.docx.pgp"

```
C:\Program Files\GNU\GnuPG>dir TextStoR.docx.pgp
 Volume in drive C is 31-072463
 Volume Serial Number is BA69-2A68

 Directory of C:\Program Files\GNU\GnuPG

25-09-2010  14:28            10.548 TextStoR.docx.pgp
               1 File(s)         10.548 bytes
               0 Dir(s)  91.132.047.360 bytes free

C:\Program Files\GNU\GnuPG>type mal.exe.gpg >> TextStoR.docx.pgp

C:\Program Files\GNU\GnuPG>dir TextStoR.docx.pgp
 Volume in drive C is 31-072463
 Volume Serial Number is BA69-2A68

 Directory of C:\Program Files\GNU\GnuPG

25-09-2010  14:47           400.048 TextStoR.docx.pgp
               1 File(s)        400.048 bytes
               0 Dir(s)  91.131.658.240 bytes free
```

[E]: 'Send' "TextStoR.docx.pgp" to D_R.

[R]: Places "TextStoR.docx.pgp" in directory and uses right mouse option 'Decrypt & Verify "TextStoR.docx.pgp"'

PGP states that message is signed by Sender, but has not only placed "TextStoR.docx" file in directory but also "mal.exe".



Of course the malicious content could also be of another type, e.g. incriminating .

Experiment #2: piggy-backing secure viewer messages

OpenPGP supports a 'secure viewer' concept: text that in principle can only be viewed in a special viewer and cannot be copied or stored. In this experiment we demonstrate that we can also add secure viewer messages. We work in the same context as in Experiment #1.

[E]: Write a message that eavesdropper E would like to appear originating from Sender in a text.txt file, for instance

****

Dear Recipient,

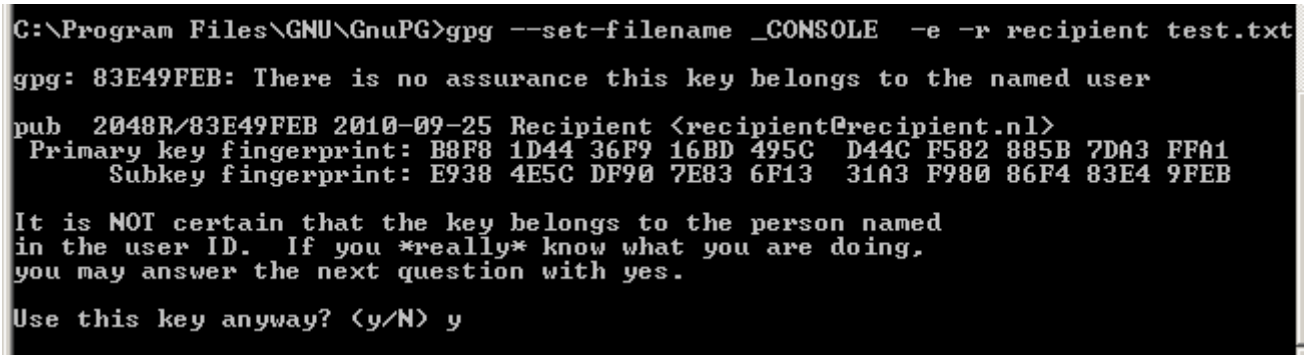
Please sell all my stock as soon as possible.

Yours sincerely,

Sender

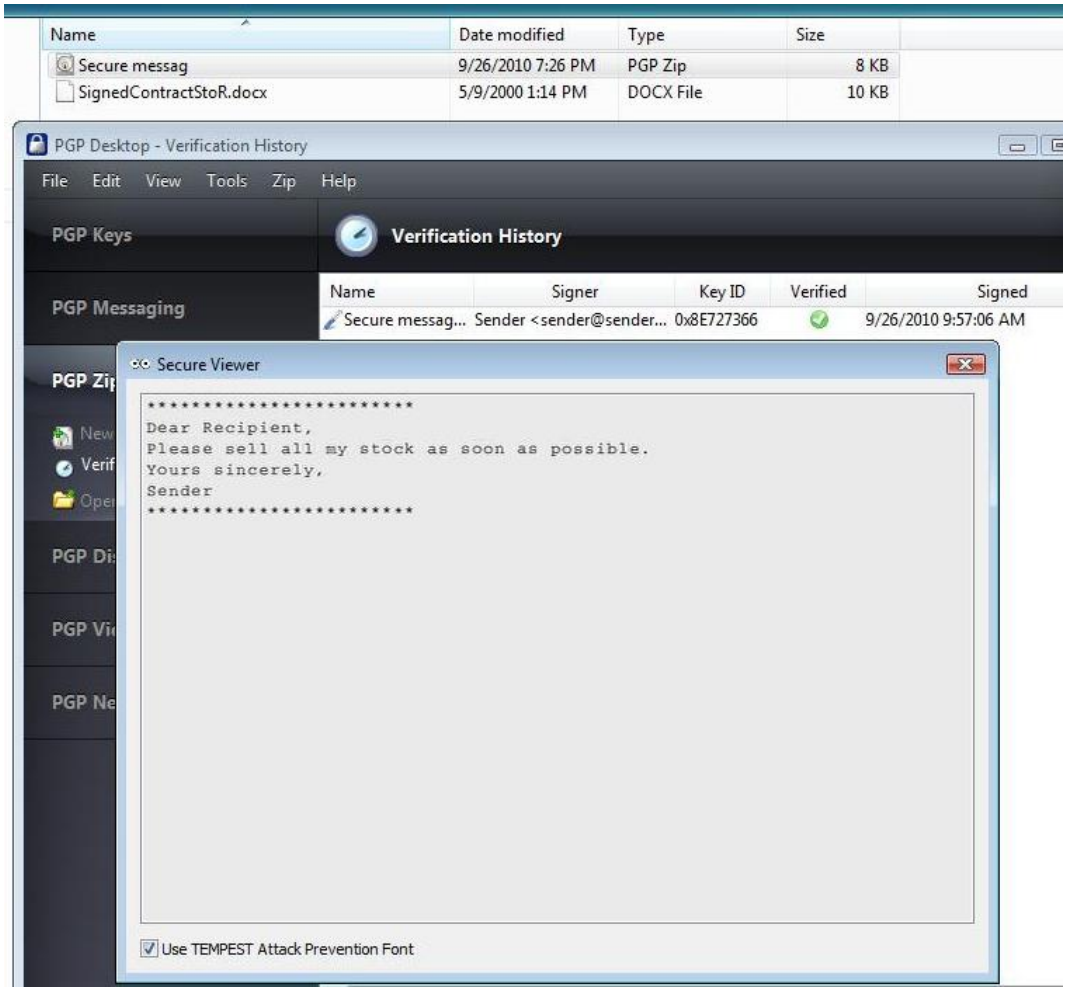
****

[E]: Encrypt text.txt file with public key of D_R stipulating usage of "secure viewer" (i.e. _CONSOLE) leading to file text.txt.gpg



[E]: Operate as in Experiment #1: concatenate text.txt.gpg to TextStoR.docx.pgp. Rename result to "secure_messag.pgp"; send the adapted file to R.

[R]: Place "secure_messag.pgp" in a directory and use the right mouse option 'Decrypt & Verify "secure_messag.pgp"'



Experiment #3: piggy-backing with non-detached signed files

[S]: Create a Word document 'SignedContractStoR.docx' pertaining a message from Sender to Recipient that is only signed by Sender (but not encrypted for Recipient).

[S]: Sign the document "SignedContractStoR.docx" with the private key of S resulting in one signed file "SignedContractStoR.docx.pgp". That is, we do not use a 'Detached Signature' and the file "SignedContractStoR.docx.pgp" contains both the original contract (in compressed form) and the signature.

[S]: 'Send' file "SignedContractStoR.docx.pgp" from D_S tot D_R but this actively intercepted by L_E. That is, this file is manipulated as indicated below and sent through to D_R afterwards.

[E]: Generate 'random PGP key' and sign malicious file "mal.exe" with this random key, resulting in file "mal.exe.pgp". Note: we do not use compression (-z 0 option in gpg) as otherwise gpgsplit is not able to do sensible packet parsing.

[E]: Run gpgsplit to decompile the result in three packets (see screen capture below).

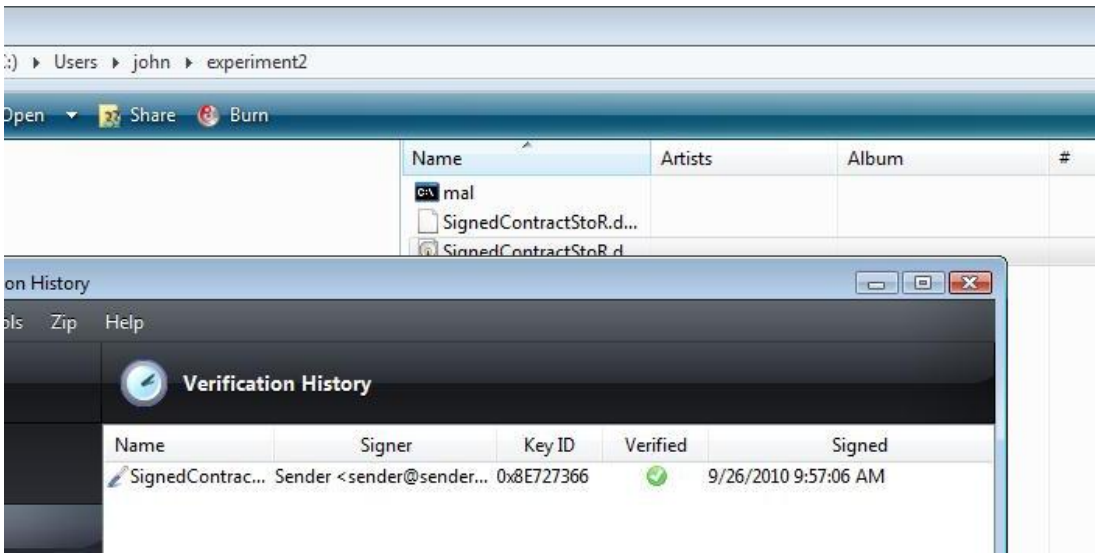
```
C:\Program Files\GNU\GnuPG>gpg -z 0 -s mal.exe

You need a passphrase to unlock the secret key for
user: "Random <random@random.nl>"
2048-bit RSA key, ID E19E05B8, created 2010-09-26


C:\Program Files\GNU\GnuPG>gpgsplit.exe -v mal.exe.gpg
gpgsplit: writing `000001-004.onepass_sig'
gpgsplit: writing `000002-011.plaintext'
gpgsplit: writing `000003-002.sig'
```

[E]: Rename "000002-11.plaintext" to file "mal_plain" (we need this file in the next experiment).

[E]: Concatenate "mal_plain" package to "SignedContractStoR.docx.pgp" and send this through to Recipient.

[R]: Place "SignedContractStoR.docx.pgp" in directory and use the right mouse option 'Decrypt & Verify "SignedContractStoR.docx.pgp"'.

PGP states that "SignedContractStoR.docx" is signed by Sender, but has not only placed "SignedContractStoR.docx" file in directory but also "mal.exe".



Experiment #4: piggy-backing with detached signed files

[S]: Create a Word document 'SignedContractStoR2.docx' pertaining a message from Sender to Recipient that is only signed by Sender (but not encrypted for Recipient).

[S]: Sign the document "SignedContractStoR2.docx" with the private key of S using a 'Detached Signature' resulting in a file "SignedContractStoR2.docx.pgp" (the detached signature).

[S]: 'Send' files "SignedContractStoR2.docx" and "SignedContractStoR2.docx.pgp" from D_S tot D_R which are intercepted by L_E.

[E]: generate ‘random PGP key’ and sign “SignedContractStoR2.docx” with this random key, resulting in file “SignedContractStoR2.docx.pgp”.

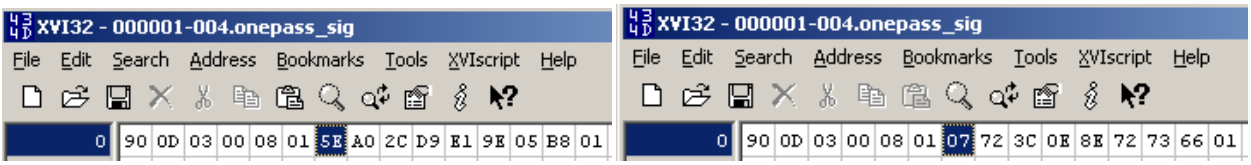
Notes:

- we do not use compression (-z 0 option in gpg) as otherwise gpgsplit is not able to do sensible packet parsing,
- with the option --force-v3-sigs we explicitly stipulate that we want version 3 signatures (GPG uses version 4 by default)
- with the option --digest-algo SHA256 we get a SHA256 based signature (GPG uses SHA-1 by default and PGP uses SHA256).

[E]: run gpgsplit to decompile the result in three packets (see the screen capture below).

```
C:\Program Files\GNU\GnuPG>gpg -z 0 --digest-algo SHA256 --force-v3-sigs -s Sign
edContractStoR2.docx

You need a passphrase to unlock the secret key for
user: "Random (random@random.nl)"
2048-bit RSA key, ID E19E05B8, created 2010-09-26


C:\Program Files\GNU\GnuPG>gpgsplit.exe -v SignedContractStoR2.docx.gpg
gpgsplit: writing `000001-004.onepass_sig'
gpgsplit: writing `000002-011.plaintext'
gpgsplit: writing `000003-002.sig'
```

[E]: Change the Key ID in the file “000001-004.onepass_sig” from that of the Random key (5EA02CD9E19E05B8) to that of the Sender key (07723C0E8E727366). Locations are specified in RFC 4880.



Note: if one would now form the file

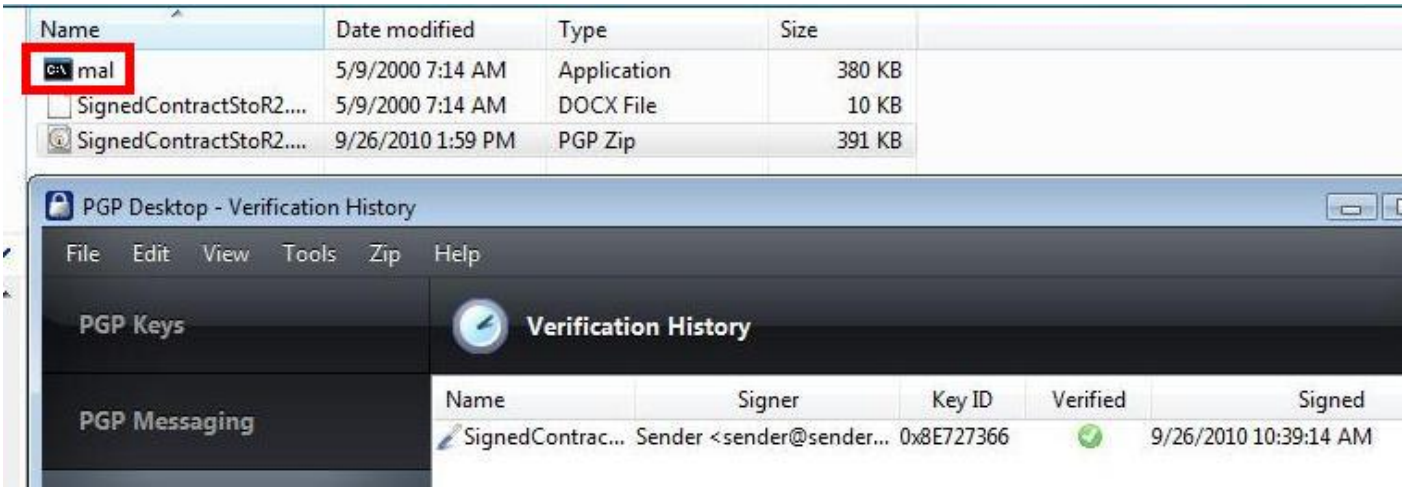
“000001-004.onepass_sig” || “000002-011.plaintext” || “SignedContractStoR2.docx.sig”

one would get one signed file similar to the one in the previous experiment, i.e. one signed file with the “SignedContractStoR2.docx” included.

[E]: E forms the file “000001-004.onepass_sig” || “000002-011.plaintext” || “SignedContractStoR2.docx.sig” || “mal_exe” where the file “mal_exe” was formed in the previous experiment. E then names this file “SignedContractStoR2.docx.pgp” and sends its through to the Recipient.

[R]: Place “SignedContractStoR2.docx.pgp” in a directory and use the right mouse option ‘Decrypt & Verify “SignedContractStoR2.docx.pgp”.

PGP states that “SignedContractStoR2.docx” is signed by Sender, but has not only placed “SignedContractStoR2.docx” file in directory but also “mal.exe”.



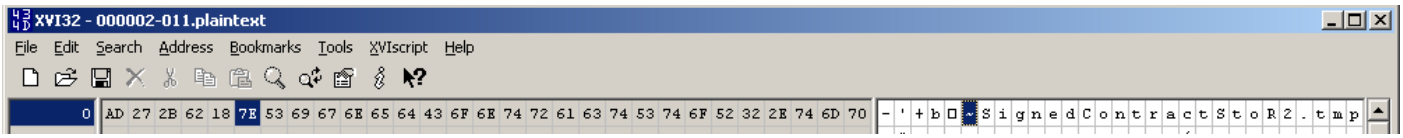
Experiment #5: piggy-backing hidden files

The OpenPGP specifications support inclusion of the filename to be used for the end result (e.g., decrypted file). This has some potential vulnerabilities (e.g., when the included file refers to an arbitrary location in the file system, e.g., c:\Windows\System32\cmd.exe). But these risks seem to be properly addressed in PGP Desktop: one seems not to be able to automatically store a file at a location different from the ciphertext file. A more fundamental vulnerability in the OpenPGP specifications is that the filename (or the length of the filename) provided by the sender is not signed, even when the file itself is signed. Moreover, PGP Desktop introduces an extra vulnerability by automatically using this filename when it is provided in the ciphertext. Together with the described piggy-back issues this introduces some interesting attack possibilities.

In the following experiment we start with a signed file (e.g. containing a contract), we cloak its filename to something that looks innocent. Then we add as plaintext a file with a changed version of that file (e.g. a changed contract). The recipient will see the signature of the legitimate file but will not get the file itself; instead it will get the changed file. That is, the recipient will trust the changed file based on the signature on the original file.

The role that Sender plays in the experiment is the same as in the previous experiment. The Eavesdropper has gained 'SignedContractStoR2.docx' and a detached signature on it. As indicated in the previous experiment Eavesdropper can reformat these into one OpenPGP message consisting of three packets One-pass (tag 4), plaintext (tag 11) and signature (tag 3).

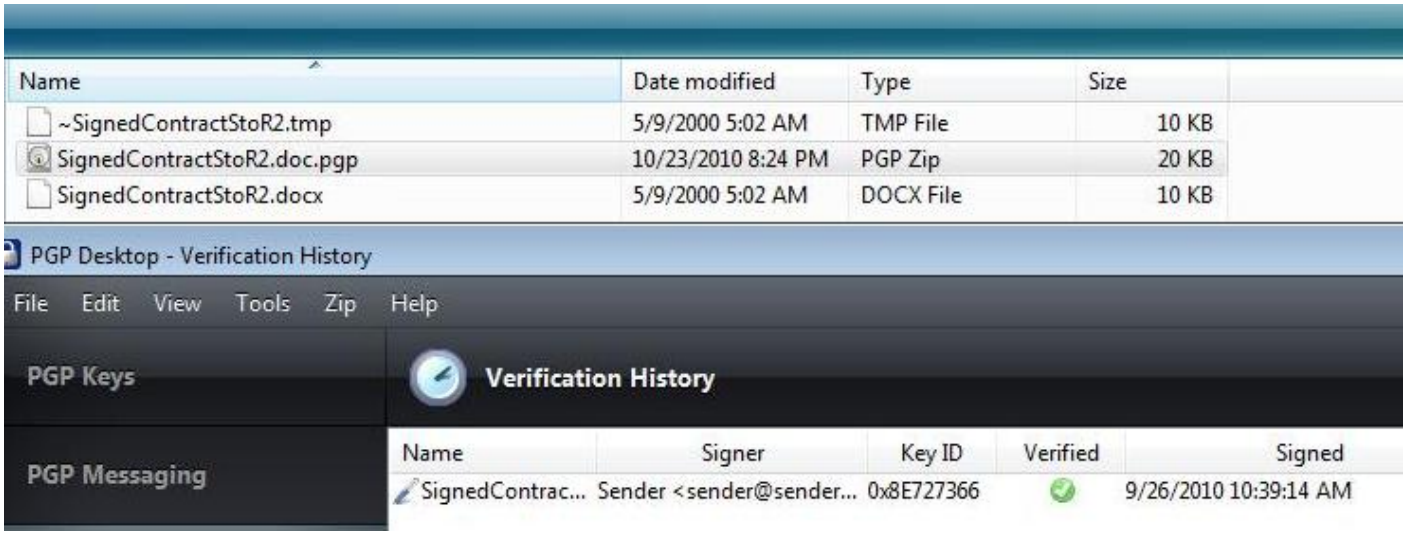
[E:] Eavesdropper changes the name in the plaintext file from "SignedContractStoR2.docx" to "~SignedContractStoR2.tmp" which we suppose is an innocent looking name. Of course another filename might be more appropriate in the specific recipient context.



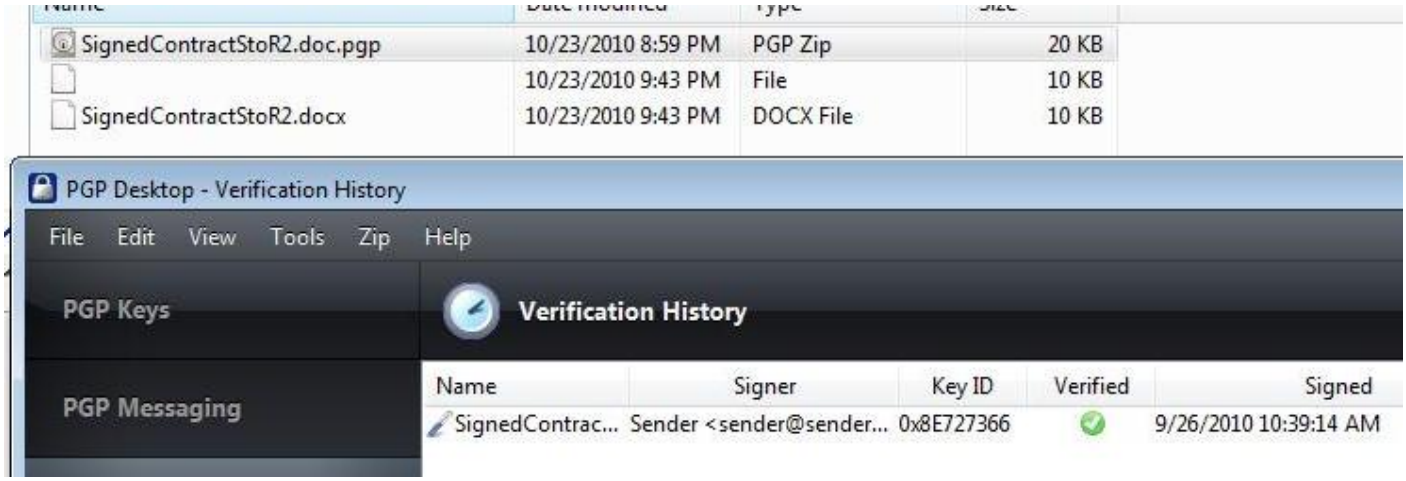
[E]: Changes the original Word document 'SignedContractStoR2.docx' to his liking and stores it in an another plaintext packet and adds it to the existing file and renames it to "SignedContractStoR2.docx.pgp". This file now consists of four OpenPGP packets: the first three packets correspond to the original (signed) file but with a changed name and the last packet corresponds to the modified Word document with the original name of the contract.

[E] Sends the file through to Recipient.

[R]: Place “SignedContractStoR2.docx.pgp” in a directory and use the right mouse option ‘Decrypt & Verify “SignedContractStoR2.docx.pgp”. PGP Desktop now states the file is signed by Sender and will store two files “~SignedContractStoR2.tmp” and “SignedContractStoR2.docx”. The first file is the legitimate file that is cloaked and that induced the valid signature; the second file is the manipulated file.



In our Vista based environment we observed that using a filename consisting of characters with ASCII code ‘7F’ in the plaintext file decrypts to a file with no visible name at all.



We remark that in this experiment we have changed the original Word document with another Word document, but one can also use a Windows ‘shortcut’ file (extension .lnk), i.e. “SignedContractStoR2.docx.lnk” and giving it the same icon as a Word document. This might lure the recipient into double clicking the ‘shortcut’ file which contains malicious content, e.g., a command line script that will be executed when the file is double clicked. The script could actually cover up its tracks and start the legitimate Word file. We also remark that in this experiment we started with a detached signature and not a with a file that contains the signature itself. We actually constructed such a file based on the detached signature. The reason for this is rather prosaic. PGP Desktop uses compressed packages for such files and gpgsplit does not really work well with such

files: it does not decompile them into lower packets. We believe it is not conceptually difficult to decompile compressed packets (tag 8).