GnuPG memory forensics

Nils Amiet and Sylvain Pelissier
nils.amiet@kudelskisecurity.com
sylvain.pelissier@kudelskisecurity.com

Kudelski Security

Abstract. After nearly 25 years of existence, GnuPG (GPG) is still a widely used solution for message encryption. GPG works with an agent (gpg-agent) containing multiple functions, including caching passphrases and encryption keys. First, this work highlights a bug of libgcrypt memory cleaning which allows reading 8 bytes of the passphrase in the clear from a memory dump. Second, it further demonstrates general techniques to retrieve passphrases and encryption keys from a memory dump, either of the gpg-agent process or a full system dump. To demonstrate our work, we provide Volatility3 plugins to retrieve associated key material and the original passphrase. We also show how this can be used as a defensive countermeasure in some practical scenarios.

1 Introduction

Pretty Good Privacy (PGP) and the open source implementation GNU Privacy Guard (GPG) are encryption solutions following the OpenPGP standard [7]. Even if GPG has been criticized in the past, it is widely used and deployed and has been publicly reviewed during many years [18]. Thus it is used in practice to protect sensitive data.

Volatility is a widely used forensics framework [11]. It is used to analyze volatile memory dump artifacts to extract data. For example, it has been used in the past to recover Bitlocker volume encryption keys while they are in RAM [17] or to solve challenges of previous SSTIC editions [4]. The framework is highly customizable and allows writing plugins in Python to fit specific needs. In 2021, the Volatility Foundation released a new version of the framework, Volatility3 [8].

This work briefly explains the basic usage of GPG, then how GPG stores the passphrases in RAM. A short description of relevant previous works is then given. From this knowledge we provide a way to extract the passphrase from a memory dump. We first show a bug of Libgcrypt (the GPG cryptographic library) memory cleaning which allows reading 8 bytes of the passphrase in cleartext. Then we show how to find AES keys in memory and how to decrypt cached items containing the passphrases.

We give practical examples where these methods may be applied and, to demonstrate our analysis, we provide Volatility3 plugins implementing our methods.

2 GPG usage of cached items

A common way to decrypt data with GPG on a command line is as following:

```
$ gpg -d clear.gpg
gpg: encrypted with 3072-bit RSA key, ID 8BEE55C2F43F1E63, created
    2021-07-14
      "user-test <user@test.org>"
Hello GPG
```

The first time the decryption is called, the system asks the user for their passphrase to decrypt the private key needed to decrypt the file. Then for the subsequent decryptions, the passphrase is not asked but read from cache. The same mechanism is used for symmetric-key encryption. The cache time to live has a default value of 10 minutes. After the time to live elapsed, the cached item is cleared from memory.

To avoid having key material directly in cleartext in memory, GPG encapsulates such key material before storing it in memory. The idea of that is that if a TPM is available, then the encapsulation key can be stored in a safe memory area. However, TPMs are usually not used and the encapsulation key stays in regular memory.

3 GPG memory structure

A cached item is stored in the `ITEM` structure. We find this structure in `gnupg/agent/cache.c` (GPG version 2.3.4):

```
/* The cache object.  */
typedef struct cache_item_s *ITEM;
struct cache_item_s {
  ITEM next;
  time_t created;
  time_t accessed;  /* Not updated for CACHE_MODE_DATA */
  int ttl;  /* max. lifetime given in seconds, -1 one means infinite
     */
  struct secret_data_s *pw;
  cache_mode_t cache_mode;
  int restricted;  /* The value of ctrl->restricted is part of the
      key.  */
  char key[1];
};
```

```

/* The cache himself.  */
static ITEM thecache;
```

Listing 1. The cache_item_s structure

The ITEM structure is a chained list containing the cached item address and additional data. Among them, there are the time of creation and time of last access. These values are unix epoch times, the number of seconds elapsed since January 1, 1970. Then there is the time to live (ttl) field which is the number of seconds gpg-agent has to keep the item in cache. By default it is set to 10 minutes (0x0258 seconds). If gpg-agent found that the last accessed time is older than the time to live, the item is cleared from cache. After that we have the address of a secret_data_s structure. The secret_data_s structure contains the length of the cached item in bytes followed by the encrypted item with AES in key wrap mode:

```
struct secret_data_s {
  int  totallen; /* This includes the padding and space for AESWRAP.
       */
  char data[1];  /* A string.  */
};
```

Listing 2. The secret_data_s structure

The encapsulation key used to encrypt the data is generated randomly when gpg-agent starts. Since GPG also stores the encapsulation key in memory, one simply needs to know where it is stored in memory to then decrypt the cached item.

3.1 AES Key wrap

GnuPG uses AES Key Wrap mode of operation to encapsulate key material in memory as defined in RFC 3394 [10]. The AES Key wrap algorithm is used to encrypt (wrap) keys or secrets with another key. It is used in several solutions like Apple FileVault 2 [6] or Cryptomator [14]. As shown in figure 1, the mode uses two 64-bit blocks concatenated together as input of AES encryptions.

This step is iterated 6 times and the final **R** values obtained are outputed as the ciphertext. Decryption works in exactly the same way but in the reverse order. The initialization vector (IV) can be any value but the RFC default value is 0xa6a6a6a6a6a6a6a6. It allows for verification of the integrity of the decrypted key after the decryption. If the last decrypted block yields a value that starts with the IV, decryption is

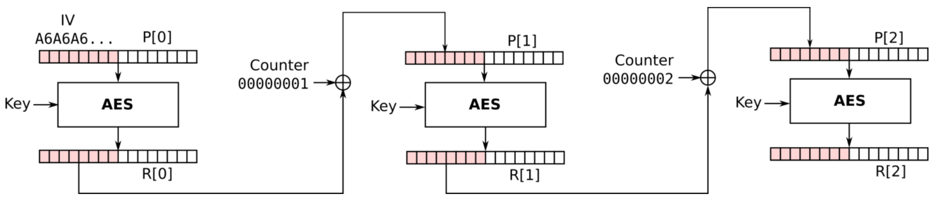

Fig. 1. First iteration of AES Key wrap encryption

considered correct as long as no other error is returned. GPG uses the implementation of AES key wrap provided by the libgcrypt library.

In GPG, the cached item encryption is used to prevent attackers from simply grepping for passphrases in memory as commented in `gnupg/agent/cache.c` (GPG version 2.3.4) and shown in listing 3.

```
/* The encryption context.  This is the only place where the
   encryption key for all cached entries is available.  It would be
       nice
   to keep this (or just the key) in some hardware device, for
       example
   a TPM.  Libgcrypt could be extended to provide such a service.
   With the current scheme it is easy to retrieve the cached entries
   if access to Libgcrypt's memory is available.  The encryption
   merely avoids grepping for clear texts in the memory.
       Nevertheless
   the encryption provides the necessary infrastructure to make it
   more secure.  */
static gcry_cipher_hd_t encryption_handle;
```

Listing 3. GPG memory threat model

However, as we will see later, someone who has access to the memory can also retrieve the encryption key and decrypt cached items anyway.

4 Previous works

A previous problem concerning `gpg-agent` and the cached items was exploited by GPG Reaper [16]. The time to live of cached items was not checked if no `gpg-agent` action was performed and thus some items may stay in memory indefinitely. Then, if a machine is compromised, the guru debug level (`--debug-level guru`) allows displaying cached items in clear if they were not deleted. The time to live problem was corrected in version 2.2.6. The guru debug level, as shown in listing 4, is still working as intended, for example, if we decrypt a file while the passphrase is still in memory.

```
$ gpg --debug-level guru -d clear.gpg
...
gpg: DBG: chan_4 -> GETINFO cmd_has_option GET_PASSPHRASE repeat
gpg: DBG: chan_4 <- OK
gpg: DBG: chan_4 -> GET_PASSPHRASE --data --repeat=0 --
    S9319569F117FE96D X X Enter+passphrase%0A
gpg: DBG: chan_4 <- D testpassword
gpg: DBG: chan_4 <- OK
...
```

Listing 4. GPG debug level guru

The passphrase `testpassword` is returned in clear from `gpg-agent`. However, access to the machine containing the cached item is required in this case.

An interesting Volatility plugin allows extracting Bitlocker volume encryption keys from memory dumps [17]. This plugin uses a known method [9, 12] which consists in scanning the memory and searching by blocks of 16 bytes if the block satisfies the AES key schedule relations with respect to the blocks next to it. If such blocks are found, we can conclude that an AES key was found in memory. Since Bitlocker uses AES in various modes for volume encryption, this technique is used to recover the volume encryption keys.

5 Cached item retrieval

Two attack vectors will be discussed here allowing to retrieve passphrases in GPG memory.

To avoid having sensitive values left in memory after processing, libgcrypt deletes those values when they are not used anymore. For example, a variable is wiped with the function `wipememory` and the stack is cleaned with the function `_gcry_burn_stack`. However, in libgcrypt, the function `_gcry_cipher_aeswrap_decrypt` (`libgcrypt/cipher/cipher-aeswrap.c` on line 81) did not clean a temporary variable containing the last decrypted block. Suppose we use GPG to decrypt some cleartext using a passphrase. At the end of the cached item decryption, the temporary variable contains the IV value `0xa6a6a6a6a6a6a6a6` followed by the first 8 bytes of the passphrase. For example, if a dump of gpg-agent's memory using `gcore` or a dump of the whole system memory using LiME [15] can be obtained, then, we should retrieve the constant `0xa6a6a6a6a6a6a6a6` in memory, next to the first 8 bytes of the passphrase. We reported this problem to GPG maintainers [13]. This was quickly corrected and following versions of GPG 2.3.4 should not be affected by this issue.

We saw that each cached entry has two timestamps `created` and `accessed` of type `time_t`. This information can be leveraged to search for such patterns in memory and retrieve the location of `cache_item_s` instances. If we can estimate the time of creation of the cached item, we can search for masked timestamps concatenated in memory followed by the time to live value. For example the regular expression `.{3}\x00\x00\x00\x00.{3}\x61\x00\x00\x00\x00\x58\x02` will search for all timestamps created after July 27, 2021 12:45:52 PM and before February 6, 2022 5:06:08 PM with a time to live of 10 minutes. To further reduce the number of false positives during the search, for each match, the created time can be checked to be less than or equal to the accessed time. Then, as soon as the `ITEM` structure has been found, we can access the `secret_data_s` structure.

To recover the encryption key, we used the same method as the one used by the Bitlocker plugin [17]. We scan the process memory until we find a 128-bit expanded AES key. Then we use this key to decrypt the cached item. If the integrity of AES key wrap is verified we know we have properly decrypted the cached item and recovered the passphrase.

6 Real-world use cases

This section describes real-world use cases where GPG is used and in-memory key material recovery has an impact. From a general point of view, if an attacker has physical access to a machine, they can copy the volatile memory and later apply the techniques explained before.

Memory forensics may be used during a criminal investigation to analyze memory dumps obtained during a search and seizure [5]. After the data has been copied the investigator may need to obtain the passphrases stored encrypted in cached items to further decrypt conversations. These techniques may also be applied to investigate virtual machines stored remotely in servers which are seized.

Ransomware is a common problem nowadays. These malicious software encrypt files on an infected machine and ask the owner for a ransom so that they can recover their files. It happens that some ransomware, rely on well studied tools to encrypt a user's data. Ransomware such as KeyBTC [1], VaultCrypt [2] or Qwerty [3] use GPG to encrypt files.

In the event that a victim of such a ransomware just noticed what happened when they get infected, the victim or the incident response team could make a memory dump of the whole system and later retrieve the password or decryption key from memory. Thus, thwarting the ransomware threat and retrieving the original files without having to pay a ransom at all. Note that the ransomware would have to rely on symmetric encryption for this to work (`gpg --symmetric`) and, so far, we have not found any ransomware relying on GPG symmetric encryption.

7 Open source contributions

We developed two plugins for Volatility3 available at `https://github.com/kudelskisecurity/volatility-gpg`. The first plugin retrieves partial (or complete, up to 8 characters) passphrases from memory by searching in `gpg-agent`'s memory the constant IV of aes-wrap. This plugin would not work on versions of GPG later than version 2.3.4. Listing 5 shows an example of usage on an Ubuntu 21.10 VM dump.

```
$ vol -f ubuntu-21.10-vm-gpg.raw -s ./volatility-gpg/symbols/ -p ../
    gpg-mem-forensics/volatility-gpg/ linux.gpg_partial
Volatility 3 Framework 2.0.0
Progress:  100.00               Stacking attempts finished
Offset  Partial GPG passphrase (max 8 chars)

0x7fb73d53a2a0  my_passp
```

Listing 5. Partial passphrase recovery

The first 8 bytes of the passphrase were found in clear in memory.

```
$ vol -f ubuntu-21.10-vm-gpg.raw -s ./volatility-gpg/symbols/ -p ../
    gpg-mem-forensics/volatility-gpg/ linux.gpg_full
Volatility 3 Framework 2.0.0
Progress:  100.00               Stacking attempts finished
Offset  Private key     Secret size     Plaintext

0x7fb738002658  0b78497b0d26239211b8841c59e943f7        32
    my_passphrase
```

Listing 6. Full passphrase recovery

The second plugin retrieves cached items in memory and cache encryption keys and therefore helps recover plaintexts. An example of usage on an Ubuntu 21.10 VM dump is shown in listing 6, where the plugin successfully found the entire passphrase `my_passphrase` in memory. The first plugin execution took 6.4 seconds and the second 59.7 seconds on an Intel Core i7-7600U CPU for a 1GB RAM dump.

8 Conclusions

To conclude, we have analyzed a bug in libgcrypt where after cleaning memory it was still possible to read 8 bytes of the passphrase in clear from a memory dump. We have further analyzed how to decrypt cached items stored in GPG memory. Our work highlights that GPG is a solid encryption solution, but it should be used in conjunction with a TPM or a secure enclave solution to harden the security against physical attacks.

References

1. Lawrence Abrams. Keybtc, a simple yet effective encrypting ransomware, 2014. [Online; accessed 10-December-2021].
2. Lawrence Abrams. Vaultcrypt uses batch files and open source gnupg to hold your files hostage, 2015. [Online; accessed 10-December-2021].
3. Lawrence Abrams. Qwerty ransomware utilizes gnupg to encrypt a victims files, 2018. [Online; accessed 10-December-2021].
4. Pierre Bienaimé. Solution du challenge SSTIC 2020. 2020.
5. Richard Carbone, C. Bean, and Martin Salois. An in-depth analysis of the cold boot attack: Can it be used for sound forensic memory acquisition? 2011.
6. Omar Choudary, Felix Gröbert, and Joachim Metz. Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption. *IACR Cryptology ePrint Archive*, 2012:374, 2012.
7. Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. OpenPGP Message Format. RFC 4880, November 2007.
8. Volatility Foundation. Volatility 3 1.0.1, 2021. [Online; accessed 26-December-2021].
9. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium (USENIX Security 08)*, San Jose, CA, July 2008. USENIX Association.
10. Russ Housley and Jim Schaad. Advanced Encryption Standard (AES) Key Wrap Algorithm. RFC 3394, October 2002.
11. Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st edition, 2014.
12. Sylvain Pelissier. In radare2, /c means Cryptography. *R2Con*, 2020.
13. Sylvain Pelissier. First 8 bytes of cache item left in clear in memory after decryption., 2021. [Online; accessed 10-December-2021].
14. Skymatic. Cryptomator, 2021. [Online; accessed 03-January-2022].
15. Joe Sylve. LiME Linux Memory Extractor, 2012. [Online; accessed 10-December-2021].
16. Kacper Szurek. GPG Reaper, 2018. [Online; accessed 22-December-2021].
17. Marcin Ulikowski. Volatility Framework: bitlocker. 2016.
18. Koch Werner. A new future for gnupg, Jan 2022.