RADICALLY OPEN SECURITY

Penetration Test Report

Summitto B.V.

V 1.0
Diemen, November 22nd, 2019
Confidential

Document Properties

Client	Summitto B.V.
Title	Penetration Test Report
Target	https://github.com/summitto/pgp-key-generation commit hash: 3a27296ac501c196e9c5948768c95f704a4b402f
Version	1.0
Pentester	Stefan Marsiske
Authors	Stefan Marsiske, Patricia Piolon
Reviewed by	Patricia Piolon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	November 19th, 2019	Stefan Marsiske	Initial draft
1.0	November 22nd, 2019	Patricia Piolon	Review

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	John Sinteur
Address	Overdiemerweg 28 1111 PP Diemen The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between November 7, 2019 and November 15, 2019, Radically Open Security B.V. carried out a code audit for Summitto B.V.

This report contains our findings as well as detailed explanations of exactly how ROS performed the code audit.

1.2 Scope of work

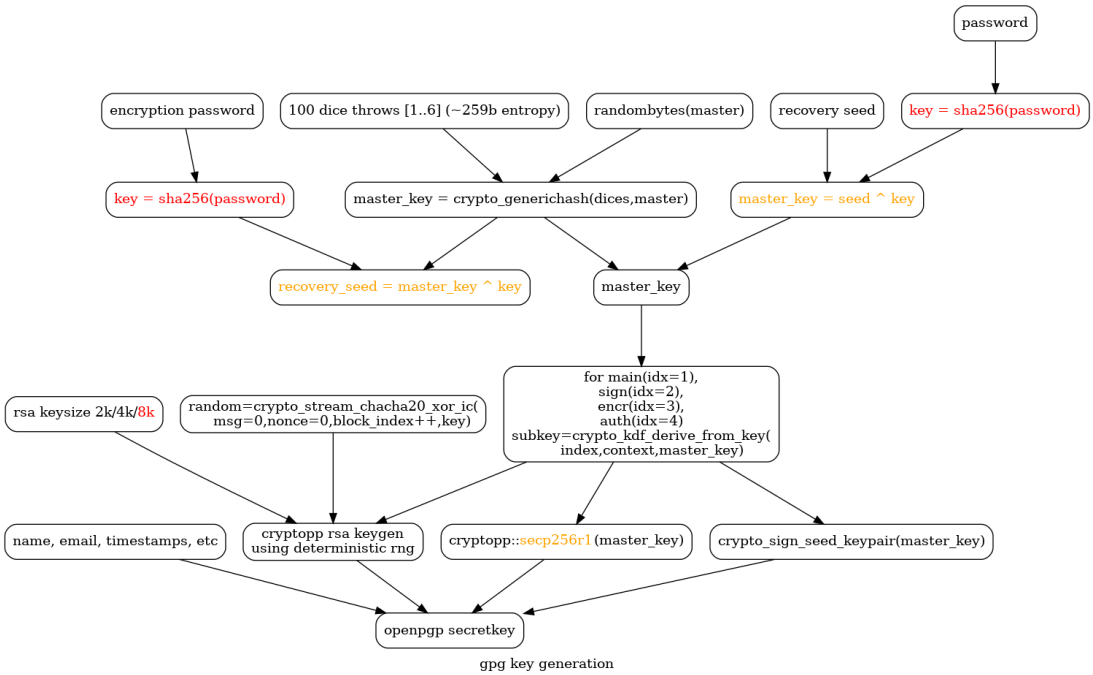
The scope of the penetration test was limited to the following target:

- https://github.com/summitto/pgp-key-generation commit hash: 3a27296ac501c196e9c5948768c95f704a4b402f

A breakdown of the scoped services can be found below:

- Code audit specified target (including reporting): 3-6 days
- **Total effort: 3 - 6 days**

A schematic of the functioning of the the target application



Function diagram of the target program

1.3 Project objectives

Since this is tool is meant to be generate cryptographic key material by the user on an air-gapped computer the focus was on info leaks and weak key generation rather than privilege escalation or remote code execution.

1.4 Timeline

The Security Audit took place between November 7, 2019 and November 15, 2019.

1.5 Results In A Nutshell

During this review we focused on weak key material being generated and sensitive data being leaked.

We found a couple of issues which would probably not cause any problems if the tool is used as intended. Operator errors can never be ruled out, however, so it makes sense to build defense in depth to limit the possibilities of such a thing happening.

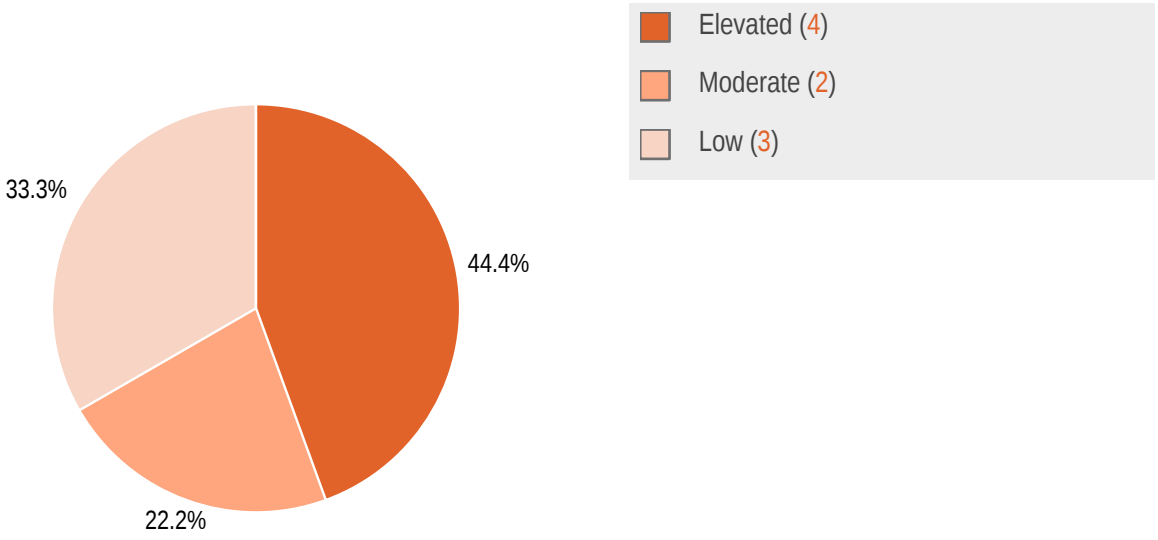
We also found two cases of invalid or insecure parameters being used, which could lead to the choosing of weak key material or cryptographic algorithms.

1.6 Summary of Findings

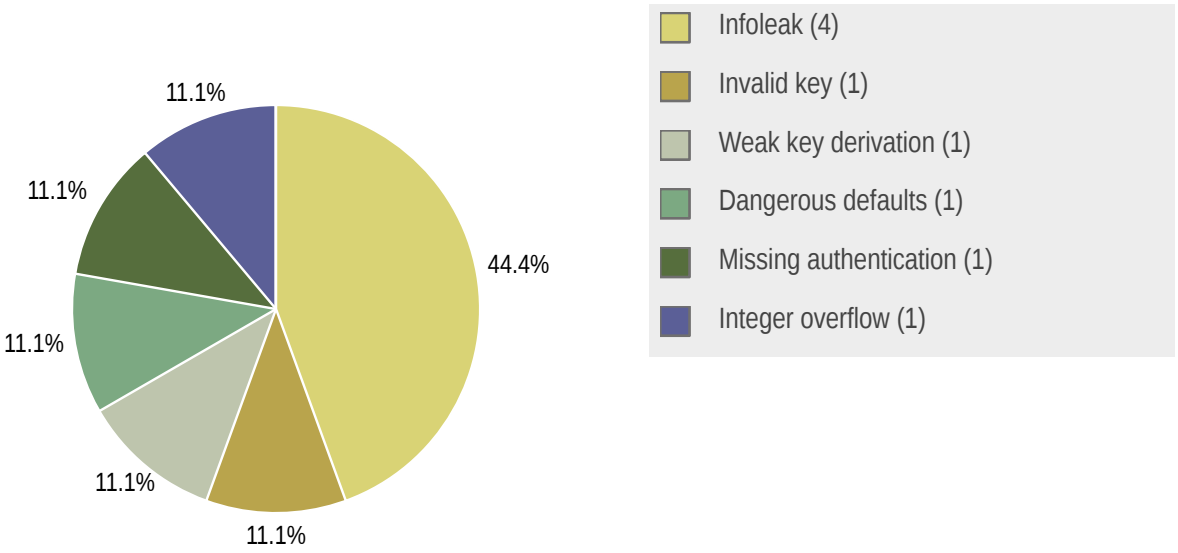
ID	Type	Description	Threat level
SMT-001	Invalid Key	The target tool allows the generation of ECDSA keys. ECDSA is a brittle algorithm which is difficult to implement properly and has some parameters that are invalid. The current implementation does not check for those invalid parameters.	Elevated
SMT-002	Infoleak	Sensitive key material is not sanitized and can leak in memory.	Elevated
SMT-003	Infoleak	The program makes no attempt to protect sensitive data from being swapped to disk.	Elevated
SMT-004	Infoleak	8192 bit keys are officially not supported in gnupg, one of the consequences of that is, that gnupg does not protect the sensitive material properly in memory and thus these keys can be swapped out to disk and leak.	Elevated
SMT-005	Weak Key Derivation	The master key in the recovery seed is only weakly protected.	Moderate
SMT-006	Dangerous Defaults	The key generated by the target application fails to specify a preferred encryption and hash algorithm in the corresponding OpenPGP packet.	Moderate

SMT-007	Missing Authentication	The master key in the recovery seed is only weakly protected.	Low
SMT-008	Infoleak	By not checking bounds on an int array access, it is theoretically possible to leak 4 bytes of memory at a time. This is especially dangerous if this function is called after processing and not sanitizing sensitive data (see issue SMT-002). However, in this case the only functions that call the affected leaky functions do check bounds, thus in the current implementation this cannot currently be classified as a serious issue.	Low
SMT-009	Integer overflow	There is an integer overflow in time_utils::days_since_unix_epoch(int year) which is harmless in the current implementation but might cause issues in other programs reading OpenPGP keys generated by this tool.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
SMT-001	Invalid Key	In general it is better to not support ECDSA keys at all (since secp256r1 is not a safe curve: https://safecurves.cr.yp.to/) or call `Validate()` on private keys after initializing them.
SMT-002	Infoleak	Sanitize sensitive memory areas after usage either with `sodium_memzero()` or rely on `sodium_munlock()` (see issue SMT-003) which also sanitizes sensitive areas. Furthermore use `sodium_stackzero()` to clear the stack before exiting the process.
SMT-003	Infoleak	Use `sodium_mlock()` and `sodium_munlock()` to protect the regions of RAM containing sensitive material.
SMT-004	Infoleak	Disallow generating 8k RSA keys.
SMT-005	Weak Key Derivation	Recommendation: derive the key using: `crypto_pwhash(salt, password)` and use seperate salts if possible for each user.
SMT-006	Dangerous Defaults	Set preferred hash to: SHA512 SHA384 SHA256 SHA224 and symmetric encryption algorithms to: TWOFISH AES256 AES192 AES
SMT-007	Missing Authentication	Calculate the recovery seed in the following way: `crypto_secretbox(recovery_seed, master_key, generichash(salt), recovery_seed_key)`
SMT-008	Infoleak	Since this is not a performance-sensitive path and it is good defensive coding practice, implement bounds checking on the month parameter instead of just documenting it as "undefined behaviour".

SMT-009	Integer overflow	Since the above calculated value `11761191` is reasonably large, it makes sense to use this as an upper limit for the year in a check, this would prevent overflow in 32 bit versions of the target, and is sufficiently large to also function as a limit in 64 bit versions of the target program.

2 Methodology

2.1 Planning

Our general approach during this code audit was as follows:

1. **Grepping**
   We attempted to identify areas of interest by grepping for memory operations: new, malloc, calloc, realloc, free.
2. **Static checks**
   We also used automated tools: flawfinder, cppcheck, infer and ikos to look for issues.
3. **Code reading**
   We read through all the code.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.
- **High**
  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**
  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**
  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**
  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Automated scans and static analysis

We did not find anything of interest through automated scans and static analysis.

As part of our active reconnaissance we used the following automated scans:

- clang analyze – https://clang-analyzer.llvm.org/
- flawfinder – https://www.dwheeler.com/flawfinder/
- cppcheck – http://cppcheck.sourceforge.net/
- ikos – https://github.com/NASA-SW-VnV/ikos/
- infer – https://fbinfer.com/

4 Findings

We identified the following issues:

4.1 SMT-001 — ECDSA Generation Doesn't Check for Invalid Parameters

Vulnerability ID: SMT-001

Vulnerability type: Invalid Key

Threat level: Elevated

Description:

The target tool allows the generation of ECDSA keys. ECDSA is a brittle algorithm which is difficult to implement properly and has some parameters that are invalid. The current implementation does not check for those invalid parameters.

Technical description:

According to the crypto++ wiki `.Validate()` should be called on the initialized private key:

```
privateKey.Initialize( ASN1::secp256k1(), x );

bool result = privateKey.Validate( prng, 3 );
if( !result ) { ... }
```

This is necessary since a secp256r1 (aka NIST p256) secret key represents any scalar modulo ℓ for `ℓ=2^256 − 432420386565659656852420866394968145599` thus any secret key with a value equal or higher than ℓ is invalid and furthermore the value zero is also invalid.

The NIST recommends, in (FIPS 186-4 Appendix B.4)[https://doi.org/10.6028/NIST.FIPS.186-4] pp. 61–64, that you either generate 320 bits uniformly at random, reduce modulo ℓ−1, and add 1; or do rejection sampling, drawing 256-bit n and starting over unless 0<n<ℓ. These methods keep the modulo bias respectively either small or nonexistent, and avoid n=0.

A simple test program that shows the difference (and importance of valid and invalid secret keys):

```
#include <cryptopp/eccrypto.h>
#include <cryptopp/oids.h>
#include <cryptopp/hex.h>
#include <cryptopp/osrng.h>
#include <iostream>

// compile with g++ -o test test.cpp -lcrypto++
int main(void) {
   std::string exp = "E4A6CFB431471CFCAE491FD566D19C87082CF9FA7722D7FA24B2B3F5669DBEFB";
```

```
    CryptoPP::HexDecoder decoder;
    decoder.Put((byte*)&exp[0], exp.size());
    decoder.MessageEnd();

    CryptoPP::Integer x;
    x.Decode(decoder, decoder.MaxRetrievable());
    CryptoPP::ECDSA<CryptoPP::ECP, void>::PrivateKey privateKey;

    privateKey.Initialize(CryptoPP::ASN1::secp256r1(), x);
    CryptoPP::AutoSeededRandomPool prng;

    std::cout << "valid: " << privateKey.Validate( prng, 3 ) << std::endl;

    exp = "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";

    decoder;
    decoder.Put((byte*)&exp[0], exp.size());
    decoder.MessageEnd();

    x.Decode(decoder, decoder.MaxRetrievable());

    privateKey.Initialize(CryptoPP::ASN1::secp256r1(), x);

    std::cout << "invalid " << privateKey.Validate( prng, 3 ) << std::endl;

    return 0;
}
```

Impact:

Weak or invalid keys might be generated.

Recommendation:

In general it is better to not support ECDSA keys at all (since secp256r1 is not a safe curve: https://safecurves.cr.yp.to/) or call `Validate()` on private keys after initializing them.

4.2 SMT-002 — Sensitive Data Is Not Sanitized

Vulnerability ID: SMT-002

Vulnerability type: Infoleak

Threat level: Elevated

Description:

Sensitive key material is not sanitized and can leak in memory.

Technical description:

An attacker could use one of the many infoleaks in the linux kernel to access the memory of the key generation process and recover key material, even after this process has exited.

Impact:

Sensitive key material can leak.

Recommendation:

Sanitize sensitive memory areas after usage either with `sodium_memzero()` or rely on `sodium_munlock()` (see issue SMT-003 (page 13)) which also sanitizes sensitive areas. Furthermore use `sodium_stackzero()` to clear the stack before exiting the process.

4.3 SMT-003 — Sensitive Data Can Be Swapped to Disk

Vulnerability ID: SMT-003

Vulnerability type: Infoleak

Threat level: Elevated

Description:

The program makes no attempt to protect sensitive data from being swapped to disk.

Technical description:

In case memory runs out, and there is virtual memory in a swap file/partition available the sensitive key material could be swapped out and be recoverable by an attacker.

Impact:

Sensitive key material (master key, derived keys) can leak to disk

Recommendation:

Use `sodium_mlock()` and `sodium_munlock()` to protect the regions of RAM containing sensitive material.

4.4 SMT-004 — 8192b RSA Keys Are Leaky in Gnupg

Vulnerability ID: SMT-004

Vulnerability type: Infoleak

Threat level: Elevated

Description:

8192 bit keys are officially not supported in gnupg, one of the consequences of that is, that gnupg does not protect the sensitive material properly in memory and thus these keys can be swapped out to disk and leak.

Technical description:

Excerpt from the latest gnupg sources `g10/gpg.c`:

```
#if SECMEM_BUFFER_SIZE >= 65536
            opt.flags.large_rsa=1;
#else
            if (configname)
              log_info("%s:%d: WARNING: gpg not built with large secure "
                         "memory buffer.  Ignoring enable-large-rsa\n",
                       configname,configlineno);
            else
              log_info("WARNING: gpg not built with large secure "
                         "memory buffer.  Ignoring --enable-large-rsa\n");
#endif /* SECMEM_BUFFER_SIZE >= 65536 */
```

Even if gnupg is built with `configure --enable-large-secmem` this only sets SECMEM_BUFFER_SIZE to 65535 which is not enough for 8192 bits. The recommended value is double, and this must be patched manually into the configure script.

There is even a FAQ item about why even 4096 bit RSA keys are "silly" https://gnupg.org/faq/gnupg-faq.html#no_default_of_rsa4096

Impact:

8192 bit RSA keys can leak to disk

Recommendation:

Disallow generating 8k RSA keys.

4.5 SMT-005 — Password Is Simply Hashed and Not Properly Derived for Encrypting Recovery Seed

Vulnerability ID: SMT-005

Vulnerability type: Weak Key Derivation

Threat level: Moderate

Description:

The master key in the recovery seed is only weakly protected.

Technical description:

The recovery seed is constructed as follows:

```
recovery_seed = master_key xor sha256(password)
```

The lack of salt for the hash operation enables an attacker to pre-compute hash values. If an attacker has the public key (or 2 RSA signed messages) of a target, he can test against that to verify if the password guess was correct. See also issue SMT-007 (page 16).

Impact:

An attacker can precompute hashes to make brute-forcing the master key easier than it needs to be.

Recommendation:

Recommendation: derive the key using: `crypto_pwhash(salt, password)` and use seperate salts if possible for each user.

4.6 SMT-006 — Generated OpenPGP Secretkey Has No Preferred Algorithms Specified

Vulnerability ID: SMT-006

Vulnerability type: Dangerous Defaults

Threat level: Moderate

Description:

The key generated by the target application fails to specify a preferred encryption and hash algorithm in the corresponding OpenPGP packet.

Technical description:

According to RFC4880 if none are specified then the defaults are assumed which are 3DES and SHA1.

Impact:

Peers PGP implementation falls back to defaults and might send cryptograms with weak hash or antique encryption algorithms.

Recommendation:

Set preferred hash to: SHA512 SHA384 SHA256 SHA224 and symmetric encryption algorithms to: TWOFISH AES256 AES192 AES

4.7 SMT-007 — Recovery Seed Is Not Authenticated Only Confidentiality Is Ensured.

Vulnerability ID: SMT-007

Vulnerability type: Missing Authentication

Threat level: Low

Description:

The master key in the recovery seed is only weakly protected.

Technical description:

The recovery seed is constructed as follows:

```
recovery_seed = master_key xor sha256(password)
```

While the xoring of the master_key does provide confidentiality, but does not provide integrity nor authenticity. In the current implementation, an attacker who can manipulate the recovery seed, can flip bits in it, which would go undetected by the user trying to recover their seed.

Impact:

An attacker can corrupt the recovery seed without this being detected.

Recommendation:

Calculate the recovery seed in the following way: `crypto_secretbox(recovery_seed, master_key, generichash(salt), recovery_seed_key)`

4.8 SMT-008 — Infoleaks in Time_utils

Vulnerability ID: SMT-008

Vulnerability type: Infoleak

Threat level: Low

Description:

By not checking bounds on an int array access, it is theoretically possible to leak 4 bytes of memory at a time. This is especially dangerous if this function is called after processing and not sanitizing sensitive data (see issue SMT-002 (page 12)). However, in this case the only functions that call the affected leaky functions do check bounds, thus in the current implementation this cannot currently be classified as a serious issue.

Technical description:

`constexpr int days_in_month(int year, int month) noexcept` and `constexpr int days_in_year_before_month(int year, int month) noexcept` from time_utils are accessing an array indexed by the `month` parameter. In both cases they reference an array on the stack that is composed of `int`s, there is no bounds checking happening. The comments of these functions mention this and state that this is undefined behaviour. This is indeed correct, but in a recent version of clang the behaviour is to return whatever is on the stack in the referenced location.

The affected function `days_in_month()` is called by `days_in_year_before_month()`, which in turn is called by `time_utils::tm_to_utc_unix_timestamp()`, which however does check bounds on the month parameter before calling.

Impact:

In the current implementation this is merely a cosmetic issue, however if any of these functions is called in the future directly without bounds checking it might leak `sizeof(int)` bytes of sensitive data.

Recommendation:

Since this is not a performance-sensitive path and it is good defensive coding practice, implement bounds checking on the month parameter instead of just documenting it as "undefined behaviour".

4.9 SMT-009 — Integer Overflow in Time_utils

Vulnerability ID: SMT-009

Vulnerability type: Integer overflow

Threat level: Low

Description:

There is an integer overflow in `time_utils::days_since_unix_epoch(int year)` which is harmless in the current implementation but might cause issues in other programs reading OpenPGP keys generated by this tool.

Technical description:

In `time_utils::days_since_unix_epoch(int year)` it is checked if year is `<1970`. If not it aborts, however there is an integer overflow in the following code:

```
int days_before_2000 = (2000 - 1970) * 365 + (2000 - 1970 + 1) / 4;
```

```
    // days_before_2000 = 10957

int days_from_2000 = (year - 2000) * 365 + (year - 2000 + 3) / 4;
    days_from_2000 -= (year - 2000 + 99) / 100;
    days_from_2000 += (year - 2000 + 399) / 400;
    days_before_2000 + days_from_2000;
```

As an example we calculate the first overflowing year value for 32 bit systems:

```
0xffffffff = (year - 2000) * 365 + (year - 2000 + 3) / 4 - ((year - 2000 + 99) / 100) + (year - 2000
 + 399) / 400 + 10957
0xffffffff = year*365 - 730000 + year/4 - 500.75 - year/100 - 20.99 + year/400 - 5.9975 + 10957
0xffffffff = year*365 + year/4- year/100 + year/400 - 719570.7375
0xffffffff = year(365 + 1/4 - 1/100 + 1/400) - 719570.7375
(0xffffffff + 719570.7375)/(365 + 0.25 - 0.01 + 0.0025) = year
year = 11761191.169531202
```

So if someone would call `days_since_unix_epoch(year)` with some year greater than `11761191`, the result would be wrong. Year `11761192` would e.g. return as a result only `345`.

By playing with the year for `time_utils::tm_to_utc_unix_timestamp()` it is possible to also overflow the end result of that function.

`time_utils::tm_to_utc_unix_timestamp()` is only called in `generate_derived_key.cpp` to calculate the timestamps for the following OpenPGP packet fields:

```
std::time_t key_creation_timestamp          =
 time_utils::tm_to_utc_unix_timestamp(*options.key_creation);
std::time_t signature_creation_timestamp    =
 time_utils::tm_to_utc_unix_timestamp(*options.signature_creation);
std::time_t signature_expiration_timestamp  =
 time_utils::tm_to_utc_unix_timestamp(*options.signature_expiration);
```

Here the year is controlled by the user himself, by specifying it as a parameter to the target application.

Impact:

No direct impact in the target application, however other programs handling keys generated by the target application might be affected and react in unexpected ways.

Recommendation:

Since the above calculated value `11761191` is reasonably large, it makes sense to use this as an upper limit for the year in a check, this would prevent overflow in 32 bit versions of the target, and is sufficiently large to also function as a limit in 64 bit versions of the target program.

5 Non-Findings

All our static analysis attempts came up empty-handed: neither clang-analyze, cppcheck, flawfinder, ikos nor infer found any issues.

6 Future Work

- **Filter out lazy-operator entropy**
  Just like passwords should not be "password", it makes sense to also disallow certain sequences of dice throws that are faked by lazy users simply pressing the "1" button a hundred times. This can be achieved simply by counting the number of different bigrams or trigrams of the dice throws and ensure that a statistically significant number of these appear in the user input.
- **Extensive fuzzing**
  It is a good practice to actually fuzz not only the core application, but also the underlying PGP packet library for extended periods of time.
- **Enlist in Coverity free software ccover scans**
  Coverity offers the possibility for free software to use their ccover static analysis tool; it is recommended to do so.
- **Consider not supporting ECDSA and RSA8192 keys**
  Both of these key types have their problems, not supporting them might be a reasonable decision.

7 Conclusion

The target software is a simple tool that tries to do only one thing; it keeps the complexity low and also does not expose itself to an adversary. The chosen language is not the simplest to audit for security issues. Having said that, the authors are competent in this language and are able to write safe code or at least recognize when they are not doing so. This is also evidenced by all our static analysis attempts coming up empty. It is however possible to develop an even more defensive mindset, as is shown by the few unimportant integer overflow and infoleak findings. It is also worth considering that the output of this tool is used by other tools, which could fail if its output is weak or incorrect.

Appendix 1 Testing team

Stefan Marsiske	Stefan runs workshops on radare2, embedded hardware, lock-picking, soldering, gnuradio/SDR, reverse-engineering, and crypto topics. In 2015 he scored in the top 10 of the Conference on Cryptographic Hardware and Embedded Systems Challenge. He has run training courses on OPSEC for journalists and NGOs, he played a lot with gnupg and is the maintainer of pysodium.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.