RADICALLY OPEN SECURITY

Code Audit Test Report

Centro de Autonomía Digital

V 1.0
Diemen, June 27th, 2019
Confidential

Document Properties

Client	Centro de Autonomía Digital
Title	Code Audit Test Report
Target	Provided source code: https://github.com/coyim/otr3 @3bc9d9c574442c5f88a941e78f96cb57cb3fa357
Version	1.0
Pentester	Stefan Marsiske
Authors	Stefan Marsiske, Patricia Piolon
Reviewed by	John Sinteur
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	June 1st, 2019	Stefan Marsiske	Initial draft
1.0	June 27th, 2019	Patricia Piolon	Proofread & finalized report

Contact

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

Name	Melanie Rieback
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 April 23, 2019 and June 1, 2019, Radically Open Security B.V. carried out a code audit for Centro de Autonomía Digital

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:

- Provided source code: https://github.com/coyim/otr3 @3bc9d9c574442c5f88a941e78f96cb57cb3fa357

1.3 Project objectives

The focus was less on RCE and language specific issues, and more on info leaks and the cryptographic security guarantees.

1.4 Timeline

The Security Audit took place between April 23 2019 and May 30, 2019.

1.5 Results In A Nutshell

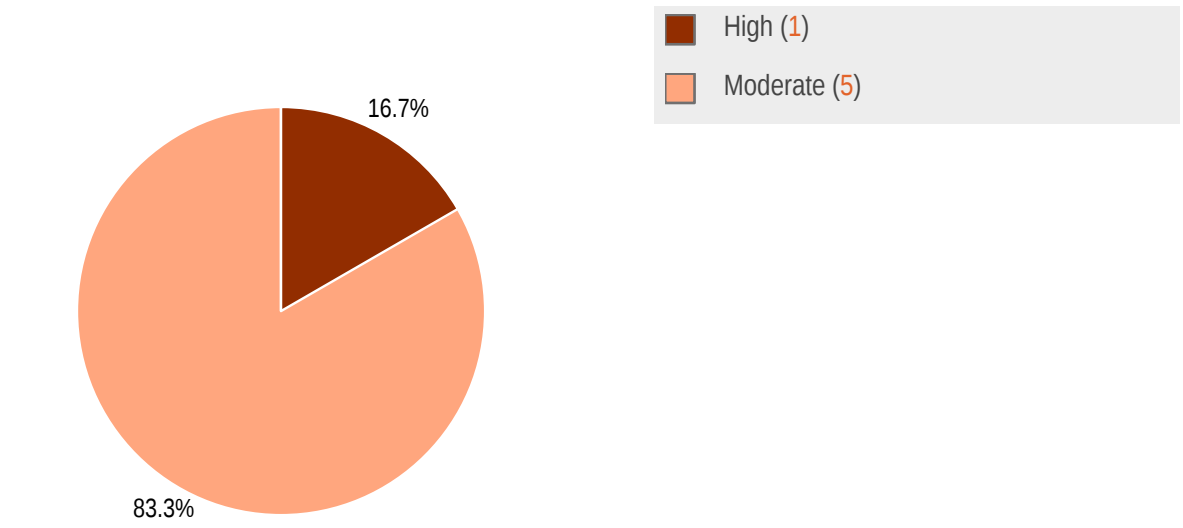
The code we inspected showed good defensive attitude with limited attack surface and even the attack surface there is seems well defended. Despite this we found a couple of issues that might leak complete key material to a local attacker. For remote attackers we verified that the time side-channel can - under certain limited conditions - be exploited to learn the Hamming weight of an ephemeral key. A remote attacker might also be able to crash the application running the OTR3 implementation.

1.6 Summary of Findings

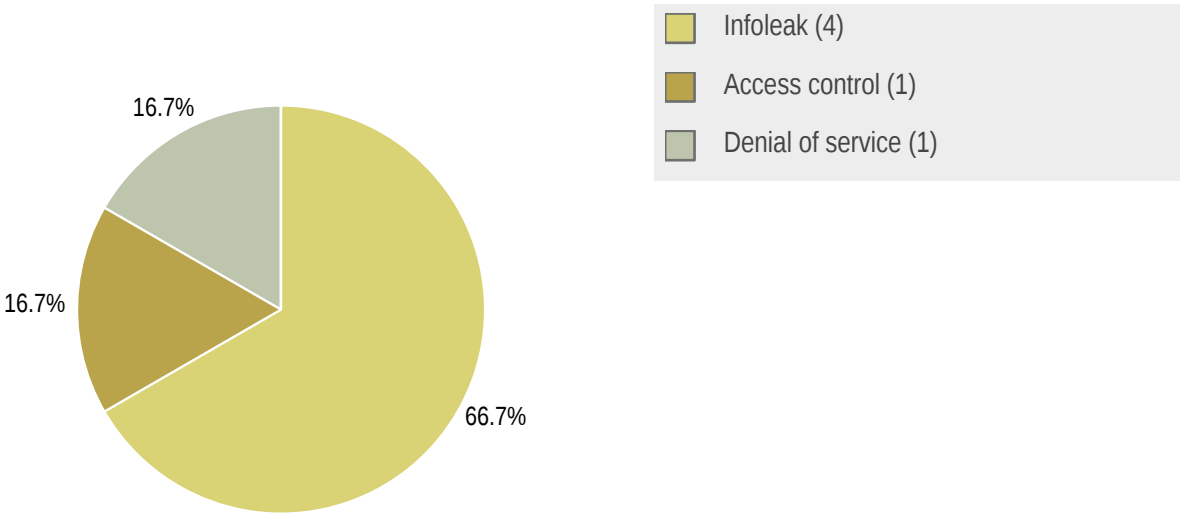
ID	Type	Description	Threat level
OTR3-001	Infoleak	Go only clears memory upon allocation, for performance reasons memory that is being freed up by an	High

		implementation is not sanitized until it is reallocated again. This means that senitive keymaterial can linger in ram not only during execution of a program handling keymaterial but also after the program finished running and being reclaimed by the OS kernel (which usually also does not sanitize memory uppon reclaiming it). The target code does attempt to sanitize the memory of sensitive keymaterial with great care. However we found a few cases were such keymaterial can still be leaked. Two relevant URLs to this question is a stackoverflow answer and a generic solution for protecting sensitive memory in Go: •https://stackoverflow.com/questions/39968084/is-it-possible-to-zero-a-golang-strings-memory-safely•https://github.com/awnumar/memguard	
OTR3-002	Infoleak	This OTR3 implementation uses a non-constant time modular exponentiation; if this is exploitable an attacker can learn the Hamming Weight of the secret key which allows to significantly reduce the search space to find the secret key.	Moderate
OTR3-003	Infoleak	Secret keys might be written to disk and escape sanitization when memory is swapped out.	Moderate
OTR3-004	Access Control	Exporting keys to disk makes them available to any local user.	Moderate
OTR3-005	Denial of Service	By sending a crafted message it is possible for a remote attacker to crash the application running OTR3.	Moderate
OTR3-006	Infoleak	AES keys are sometimes copied without wiping them after usage causing a possible local leak in RAM of them.	Moderate

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
OTR3-001	Infoleak	Our advice is to not only sanitize the AES key used when constructing the cipher, but also sanitize the aesCipher structure after usage.
OTR3-002	Infoleak	Use a constant-time implementation for modular exponentiation.
OTR3-003	Infoleak	Our recommendation is to use `mlock()` on any sensitive key-material prohibiting these to be written to disk, or alternatively consider a generic solution like https://github.com/awnumar/memguard.
OTR3-004	Access Control	Set the access rights before saving.
OTR3-005	Denial of Service	Check for the length of the R parameter before copying it.
OTR3-006	Infoleak	Do not copy unnecessarily AES keys, and wipe them after usage.

2 Methodology

2.1 Planning

Our general approach during this code audit was as follows:

1. **Code reading**
   We read through the protocol specification and all the code to identify logic bugs and other areas of interests that we investigated deeper.
2. **Static checks**
   We also used two automated tools, gosec and golangci-lint, to look for issues. Only false positives and security irrelevant findings were reported.

2.2 Risk Classification

Throughout the document, vulnerabilities or risks are labeled and categorized as:

- **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.

Please note that this risk rating system was taken from the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting.

3 Static Analysis

Through automated scans we tried to get a first feel for the code quality.

3.1 Automated Scans

We ran two static analysis tools gosec and golangci-lint, gosec did not find anything of relevance to security, golangci-lint finds were similarly irrelevant to our audit. One finding might be of interest for general code quality:

```
query.go:74:9: copylocks: QueryMessage passes lock by value: github.com/coyim/otr3.Conversation
 contains github.com/coyim/otr3.resendContext contains struct{m []github.com/coyim/
otr3.messageToResend; sync.RWMutex} (govet)
func (c Conversation) QueryMessage() ValidMessage {
```

To reproduce (and possibly integrate into a CI infrastructure) you can invoke gosec on the target sources as follows:

```
GOPATH=~/.go go get github.com/securego/gosec/cmd/gosec/...
GOPATH=~/.go go get github.com/coyim/otr3
GOPATH=~/.go ~/.go/bin/gosec ~/.go/src/github.com/coyim/otr3/...
```

Similarly golangci-lint can be invoked as follows:

```
GOPATH=~/.go go get github.com/golangci/golangci-lint/cmd/golangci-lint
GOPATH=~/.go go get github.com/coyim/otr3
cd ~/.go/src/github.com/coyim/otr3/
GOPATH=~/.go ~/.go/bin/golangci-lint run
```

4 Findings

We have identified the following issues:

4.1 OTR3-001 — AES Keys Expanded for Round Keys Are Not Sanitized by the Underlying AES Implementation.

Vulnerability ID: OTR3-001	**Retest status:** Resolved
Vulnerability type: Infoleak	
Threat level: High	

Description:

Go only clears memory upon allocation, for performance reasons memory that is being freed up by an implementation is not sanitized until it is reallocated again. This means that senitive keymaterial can linger in ram not only during execution of a program handling keymaterial but also after the program finished running and being reclaimed by the OS kernel (which usually also does not sanitize memory uppon reclaiming it). The target code does attempt to sanitize the memory of sensitive keymaterial with great care. However we found a few cases were such keymaterial can still be leaked.

Two relevant URLs to this question is a stackoverflow answer and a generic solution for protecting sensitive memory in Go:

- https://stackoverflow.com/questions/39968084/is-it-possible-to-zero-a-golang-strings-memory-safely
- https://github.com/awnumar/memguard

Technical description:

The primary code tries to diligently wipe any sensitive key material after usage, however the go implementation of AES expands the round keys into RAM and does not sanitize them after usage. This means they these AES keys could leak in various scenarios due to unnecessary big windows of exposure.

Similarly to the generic implementation of AES also architecture specific assembler implementations expand the round keys into RAM, as an example the generic implementation from: https://golang.org/src/crypto/aes/cipher.go#43 is shown below:

```
// newCipherGeneric creates and returns a new cipher.Block
// implemented in pure Go.
func newCipherGeneric(key []byte) (cipher.Block, error) {
    n := len(key) + 28
    c := aesCipher{make([]uint32, n), make([]uint32, n)}
    expandKeyGo(key, c.enc, c.dec)
```

```
    return &c, nil
}
```

Particularly the `aesCipher` structure `enc` and `dec` members contain the expanded keys. These are particularly easy to identify for example during a cold boot attack by looking for high entropy blocks of memory and then check whether the following bytes match those generated by running the AES round key expansion function. Similarly it might be possible to force the OS to swap out (see OTR3-003 (page 17)) the memory containing the AES keys and then recovering them from the disk without running risk of bits decaying as in a cold boot attack is usually the case.

Impact:

Short-term message keys can be leaked

Recommendation:

Our advice is to not only sanitize the AES key used when constructing the cipher, but also sanitize the aesCipher structure after usage.

Retest update:

The basic fix here was to try to clean up the aesCipher structure. However, since this structure has private members, the authors ended up using unsafe pointers to be able to try to wipe the content. The primary commit that fixes this issue can be seen in https://github.com/coyim/otr3/commit/399e9bf2defe197fcd31195b915240b17ea6c161. It does introduce some usages of unsafe, but in a restricted way that should be safe.

4.2 OTR3-002 — EXP Is Not Side-Channel Resistant.

Vulnerability ID: OTR3-002	**Retest status:** Resolved
Vulnerability type: Infoleak	
Threat level: Moderate	

Description:

This OTR3 implementation uses a non-constant time modular exponentiation; if this is exploitable an attacker can learn the Hamming Weight of the secret key which allows to significantly reduce the search space to find the secret key.

Technical description:

The file `otr3/sexp/bignum.go` uses `math/big` `func (*Int) Exp()`, which according to https://golang.org/pkg/math/big/#Int.Exp:

Modular exponentation of inputs of a particular size is not a cryptographically constant-time operation.

To be more specific https://golang.org/src/math/big/int.go#L455 uses `res = nat.expNN(x,y,m)` which is not timing safe. See https://golang.org/src/math/big/nat.go#L1023

This is a known problem and there is an open github issue for golang; see *"proposal: math/big: support for constant-time arithmetic"* at https://github.com/golang/go/issues/20654

There's a few tools available for investigating this:

- https://github.com/0x64616E69656C/ctgrind - updated fork of agls ctgrind - needs instrumentation by calling a dummy function which is intercepted by valgrind
- http://www.reparaz.net/oscar/misc/dudect.html - https://github.com/oreparaz/dudect - ported to work with go: https://github.com/ansemjo/dudect - associated paper: https://eprint.iacr.org/2016/1123.pdf
- a prover-based approach https://github.com/niekbouman/verifying-constant-time
- And flowtracker, for which the source seems unavailable http://cuda.dcc.ufmg.br/flowtracker/

We tried to trace where sensitive keys are being used as exponents and found three locations and the following call paths:

- `keys.Import()`: this function reads a libotr-sexp-style key file, and is not under control of an attacker. If it would be the attacker has the keys to the kingdom. Thus we ignore this path for our side-channel analysis
- `key_management.generateNewDHKeyPair()` which is called from two places:
  - `key_management.rotateOurKeys()` which in turn is called by `key_management.rotateKeys()` and finally this is called by `data_message.processDataMessageWithRawErrors()`
  - `auth_state_machine.akeHasFinished()` which is being called by both:
    - `auth_state_machine.receiveRevealSigMessage()`
    - `auth_state_machine.receiveSigMessage()`

    both `receiveRevealSigMessage()` and `receiveSigMessage()` are called by `auth_state_machine.processAKE()` which is called by `receive.receiveAKEMessage()`

- key_management.calculateDHSessionKeys() is called from both:
- `data_message.genDataMsgWithFlag()` is initiated when sending a message, an adversary needs to trigger a send, and there is little chance to abort as soon as possible after the `Exp()` has been called.

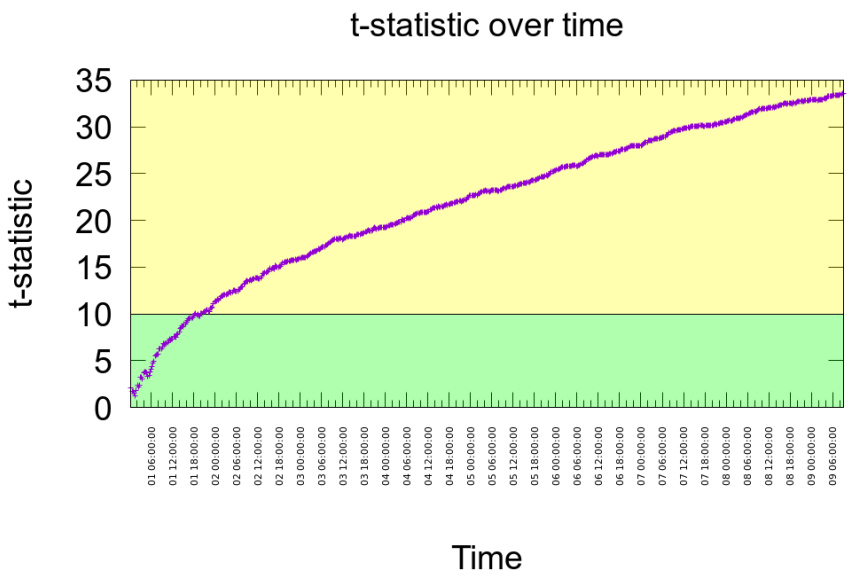
- `data_message.processDataMessageWithRawErrors()` is invoked when receiving a message and immediately after calculating the DH session keys invokes: `dataMessage.checkSign(sessionKeys.receivingMACKey, header, c.version)` which gives an attacker a means to abort further calculations by making sure the signature does not verify.

Of these 3 locations the first one was out of scope, the second one seemed more complicated to trigger so we focused on the 3rd one specifically the one triggered by `data_message.processDataMessageWithRawErrors()` and used dudect to estimate how difficult it would be to exploit this timing side-channel.

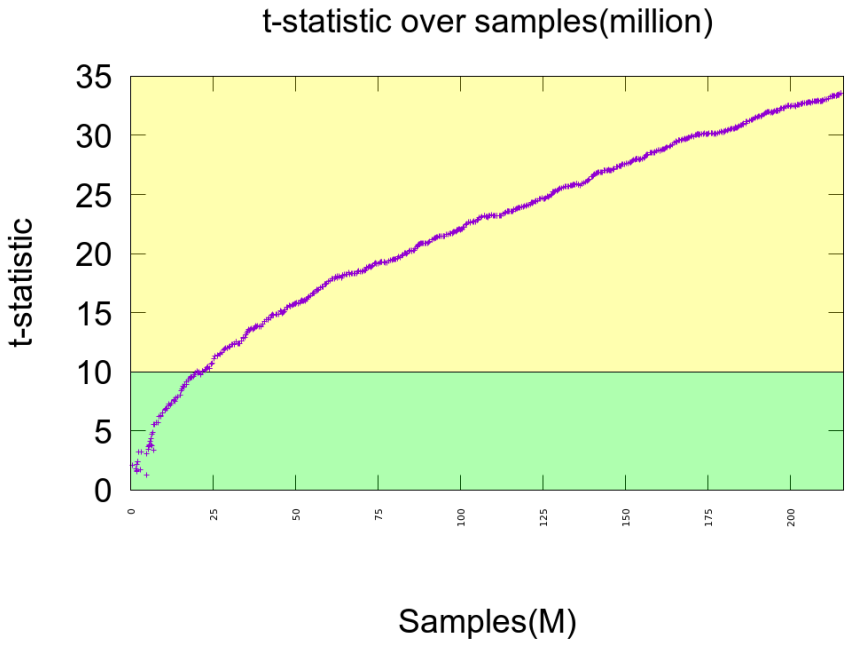
We cloned a fork of dudect which had already a golang target as an example: https://github.com/ansemjo/dudect - we then wrote our own test and then we let run it for multiple days:

```
GOPATH=~/.go make go && {echo start ; ./dudect_go_-O2 ; } 2>&1 | /usr/bin/ts -s
```

The source for the test is copied below.



t-statistic vs time



t-statistic vs samples

The results show us that with this particular test it takes about 21 million samples to reach a t-statistic that shows a definite leak, generating these 21 million samples takes about 20 hours.

Furthermore we need to note that this test was running in a single process, in a real world attack we need to also account for network communication overhead making this attack even slower.

The timing side channel really gives us only an estimation of how many bits in the secret exponent are 1, the hamming weight of the secret key. So this means after collecting enough samples the attacker still has to find out what the secret key is, extending the time needed to recover just one key. But this can be done offline. Considering that this secret key is updated regularly during a regular OTR chat session only the timespan to collect samples might be very short. However between chatsessions it is possible for an attacker to recover the key for the next first few messages during an upcomming chat session.

Source code of the test:

```
package main

// #include <stdlib.h>
// #include <stdint.h>
import "C"
import (
    "runtime"
    "crypto/rand"
   "math/big"
   //"bytes"
   "os"
   "fmt"
    "github.com/coyim/gotrax"
)

// if you change these values, change them in dut_go.c aswell!
```

```
const chunksize = 748
const measurements = 1e6

// either 0x00 or 0x01
func randombit() byte {
    b := make([]byte, 1)
    _, err := rand.Read(b)
    if err != nil {
        panic(err)
    }
    return b[0] & 0x01
}

//export init_dut
func init_dut() {}

var (
   bob *Conversation
   m []byte // 506 bytes long
)

//export prepare_inputs
func prepare_inputs(inputptr *C.uint8_t, classesptr *C.uint8_t) {
    // create slice abstractions
    allinputs := makeslice(inputptr, chunksize*measurements)
    classes := makeslice(classesptr, measurements)
    inputs := make([][]byte, measurements)
    for i := range inputs {
        inputs[i] = allinputs[i*chunksize : (i+1)*chunksize]
    }

   // set up receiver (bob)
    bob = newConversation(otrV3{}, rand.Reader)
    bob.msgState = encrypted
    bob.Policies.add(allowV3)
    plain := plainDataMsg{
        message: []byte("pwned"),
    }

    var senderKeyID uint32 = 1
    var recipientKeyID uint32 = 1

    bob.keys = keyManagementContext{
        ourKeyID:   senderKeyID + 1,
        theirKeyID: recipientKeyID + 1,
        theirCurrentDHPubKey:  fixedGX(),
        theirPreviousDHPubKey: fixedGX(),
    }

   // setup sender (alice)
    conv := newConversation(otrV3{}, rand.Reader)

   // alice key should be fixed
    ax := fixedX()
   ay := fixedGX()

    h, _ := conv.messageHeader(msgTypeData)
    m := dataMsg{
        senderKeyID:    senderKeyID,
        recipientKeyID: recipientKeyID,
```

```
        y:          ay, //this is alices current Pub
        topHalfCtr: [8]byte{0, 0, 0, 0, 0, 0, 0, 2},
    }

  // signing with wrong signingmackey so the decryption aborts early
    zeroes32 := make([]byte, 32)
  m.sign(zeroes32, h, conv.version)

  var bx, by *big.Int
  // pre-calc some test-vectors
  //b0 :=
bnFromHex("8000000000000000000000000000000000000000000000000000000000000000000000000000000000000000")
  //b0g := modExp(g1, b0)
  bf :=
bnFromHex("ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff")
  bfg := modExp(g1, bf)

    for i := range inputs {
        classes[i] = randombit()
        if classes[i] == 1 {
            bx, _ = randSizedMPI(rand.Reader, 40)
            by = modExp(g1, bx)
         //bx = b0
         //by = b0g
        } else {
         //if randombit() == 0 {
         //   bx = b0
         //   by = b0g
         //} else {
            bx = bf
            by = bfg
         //}
        }
      keys := calculateDHSessionKeys(ax, ay, by, conv.version)

      m.encryptedMsg = plain.encrypt(keys.sendingAESKey[:], m.topHalfCtr)

      if len(inputs[i]) != 748 {
         fmt.Printf("inputs[%d] len: %d\n", i, len(inputs[i]))
         os.Exit(1)
      }
      msg := append(h, m.serialize(conv.version)...)
      if len(msg) > 508 {
         fmt.Printf("msglen: %d\n", len(msg))
         os.Exit(1)
      }
      priv := gotrax.AppendMPI(nil, bx)
      if len(priv) > 44 {
         fmt.Printf("privlen: %d\n", len(priv))
         os.Exit(1)
      }
      pub := gotrax.AppendMPI(nil, by)
      if len(pub) > 196 {
         fmt.Printf("publen: %d\n", len(pub))
         os.Exit(1)
      }
      copy(inputs[i][0:508], msg)
      copy(inputs[i][508:508+44], priv)
      copy(inputs[i][508+44:508+44+196], pub)
    }
    // run GC now in an attempt to not have
```

```
    // it run during computations too much
    runtime.GC()
}

//export do_one_computation
func do_one_computation(dataptr *C.uint8_t) C.uint8_t {
    data := makeslice(dataptr, chunksize)
   _, bob.keys.ourCurrentDHKeys.priv, _ = gotrax.ExtractMPI(data[508:508+44])
    _, bob.keys.ourCurrentDHKeys.pub, _ = gotrax.ExtractMPI(data[508+44:])
    _, bob.keys.ourPreviousDHKeys.priv, _ = gotrax.ExtractMPI(data[508:508+44])
    _, bob.keys.ourPreviousDHKeys.pub, _ = gotrax.ExtractMPI(data[508+44:])
   bob.receiveDecoded(data[:508])
   return (C.uint8_t)(uint8(1))
}

func main() {}
```

Impact:

It might be possible to learn the hamming weight of ephemeral keys.

Recommendation:

Use a constant-time implementation for modular exponentiation.

Retest update:

This issue has been addressed by the introduction of http://github.com/coyim/constbn

4.3 OTR3-003 — Sensitive Data Is Not mlocked

Vulnerability ID: OTR3-003	**Retest status:** Resolved
Vulnerability type: Infoleak	
Threat level: Moderate	

Description:

Secret keys might be written to disk and escape sanitization when memory is swapped out.

Technical description:

It might be possible for a local attacker to exhaust memory while sensitive key-material is in use in RAM. Since keys are not protected with `mlock()`, these keys might be swapped out to disk and thus may leak semi-permanently. Unfortunately these keys written to disk will not be sanitized on disk when they are wiped after usage.

Impact:

Complete keys might be persisted to swap on disk.

Recommendation:

Our recommendation is to use `mlock()` on any sensitive key-material prohibiting these to be written to disk, or alternatively consider a generic solution like https://github.com/awnumar/memguard.

Retest update:

The solution to this issue was to try to make a best effort to lock secret keys. The primary commits to fix this problem are:

- https://github.com/coyim/otr3/commit/676be69c78ee3a2ed53242b1f1a7cfeeadea5d7d
- https://github.com/coyim/otr3/commit/3865303d9c2609f15622ab71edb379d78ab95672

4.4 OTR3-004 — Missing File System Access Control Settings

Vulnerability ID: OTR3-004	**Retest status:** Resolved
Vulnerability type: Access Control	
Threat level: Moderate	

Description:

Exporting keys to disk makes them available to any local user.

Technical description:

The function `ExportKeysToFile()` from `otr3/keys.go` saves keys to a file however it does not restrict access with possible access controls provided by the file system and operating system.

Impact:

Long-term keys might leak to disk and be accessible to a local attacker.

Recommendation:

Set the access rights before saving.

Retest update:

The solution to this problem was to simply create a file with more restrictive access controls: https://github.com/coyim/otr3/commit/b834494487dd1d9351993e489bddd3ff531dfa50

4.5 OTR3-005 — Crashing of the OTR3 Process by Sending a Crafted Message

Vulnerability ID: OTR3-005	**Retest status:** Resolved
Vulnerability type: Denial of Service	
Threat level: Moderate	

Description:

By sending a crafted message it is possible for a remote attacker to crash the application running OTR3.

Technical description:

It is possible to crash the process by crafting a message that has an `r` value bigger than 16 bytes. In `messages.go` the following function deserializes adversarial input:

```
func (c *revealSig) deserialize(msg []byte, v otrVersion) error {
    in, r, ok1 := gotrax.ExtractData(msg)
    macSig, encryptedSig, ok2 := gotrax.ExtractData(in)
    if !ok1 || !ok2 || len(macSig) != v.truncateLength() {
        return newOtrError("corrupt reveal signature message")
    }

    copy(c.r[:], r)
```

If the length of `r` is bigger than 16 bytes then it crashes the process, since `revealSig` as `r` defined as such:

```
type revealSig struct {
   // TODO: why this number here?
```

```
r            [16]byte
```

Impact:

Denial of using the application using the OTR3 implementation.

Recommendation:

Check for the length of the R parameter before copying it.

Retest update:

The authors solved this problem by simply checking the size of the `r` parameter during deserialization: https://github.com/coyim/otr3/commit/0f51ef999dda1716692312b0f5ac205160f88bd5

The authors noticed that the above commit introduces a very small window for timing side channels. It was fixed here: https://github.com/coyim/otr3/commit/240c8939c9d8627c7aa2831b69409fdbba13b40b

4.6 OTR3-006 — Leak of AES Key

Vulnerability ID: OTR3-006	**Retest status:** Resolved
Vulnerability type: Infoleak	
Threat level: Moderate	

Description:

AES keys are sometimes copied without wiping them after usage causing a possible local leak in RAM of them.

Technical description:

For some reason `counterEncipher(key[:], iv[:], data, dst)` in `messages.go` is being passed the AES key as a copy, which then is not wiped and thus can possibly be recovered.

Recommendation: do not make a copy of the key and preferably wipe the AES key after usage.

Impact:

AES keys can leak into RAM and can thus leak to local attacker.

Recommendation:

Do not copy unnecessarily AES keys, and wipe them after usage.

Retest update:

The authors found several places where this problem happened. It's possible the following commits are not all of them, but should cover most of them:

- https://github.com/coyim/otr3/commit/1622e417e407bd2f75efb99788e3ca656a5bdd93
- https://github.com/coyim/otr3/commit/3865303d9c2609f15622ab71edb379d78ab95672
- https://github.com/coyim/otr3/commit/5d200acad716cdcd51772bf4c81495892a7b0572

5 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

5.1 NF-001 — ModExp Leak Secrets in RAM

After finding OTR3-001 we also checked if the modular exponentiation leaks the secret exponent in RAM due to being copied temporarily, but we found that the go implementation does operate directly on the exponent and does not make any copies of it.

- Exp() https://golang.org/src/math/big/int.go#L455 calls
  - nat.expNN() https://golang.org/src/math/big/nat.go#L1023 calls
  - nat.expNNMontgomery() https://golang.org/src/math/big/nat.go#L1189 and
  - nat.expNNWindowed() https://golang.org/src/math/big/nat.go#L1125

6 Future Work

There is very little future work besides addressing the found issues.

7 Conclusion

The code looks mature and has a good defensive stance, we are looking forward to use also OTRv4 from the same authors and expect the review of that implementation to be as unrewarding as this one.

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.
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.