Replay Attack Defense of V2Ray

Junchen Bao, Mingjie Zhu, Sicheng Wang, Yinliang Wang, Yuxiao Ran

Part 1: GFW & VPN

The Great Firewall(GFW)

- A special Firewall in China
- Block access to selected foreign websites
- Control sensitive content

Virtual Private Network(VPN)

- send and receive data across shared or public networks
- External network nodes - bypass the Great Firewall

Part 1: GFW & VPN

Shadowsocks(SS)

- A free and open-source encryption protocol
- Client software to help connect to a third-party SOCKS5 proxy
- Stop updating in 2015
- Can be easily intercepted by GFW now

Part 1: GFW & VPN

V2Ray

- provide users with the software to deploy specific network environments privately
- access content that would otherwise be blocked
- Main protocol is VMess
- Most common now

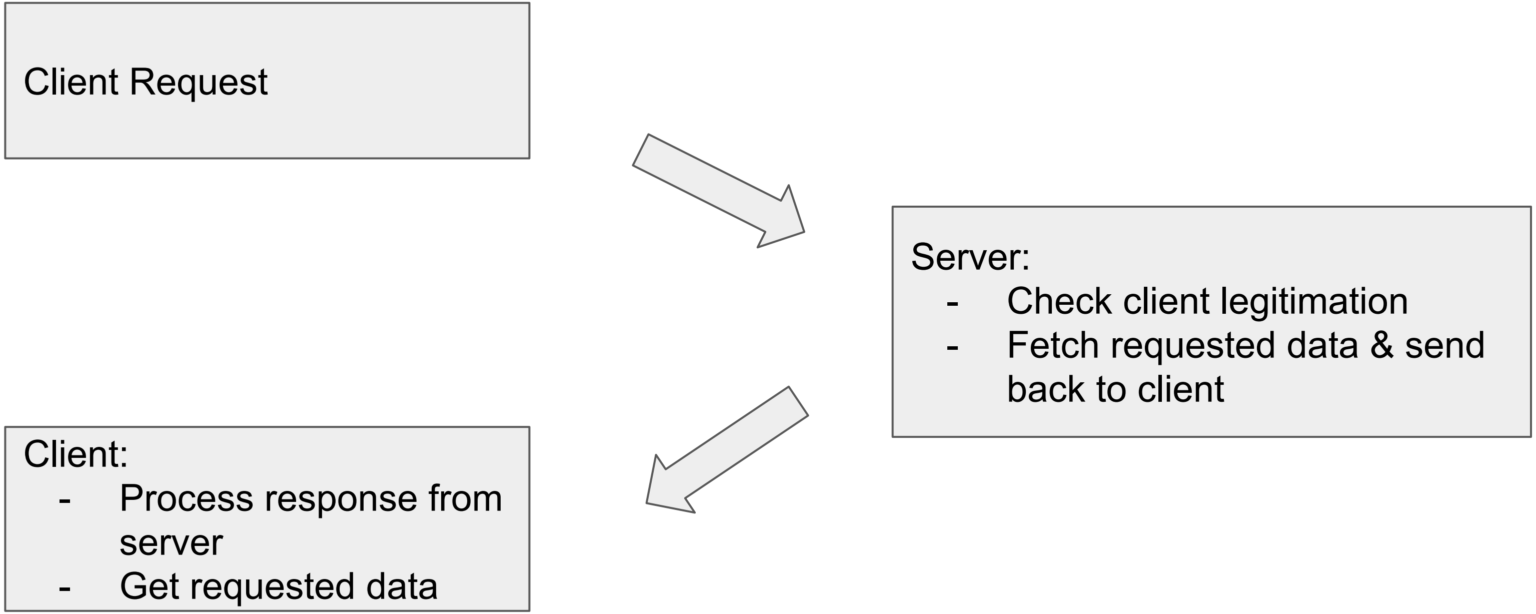
Part 2: VMess Protocol

An original encrypted communication protocol of V2Ray:

- Stateless : directly send/recv data without handshake first

- Asymmetric: different formats for client request and server response

Part 2: VMess Protocol



Client Request

Server:

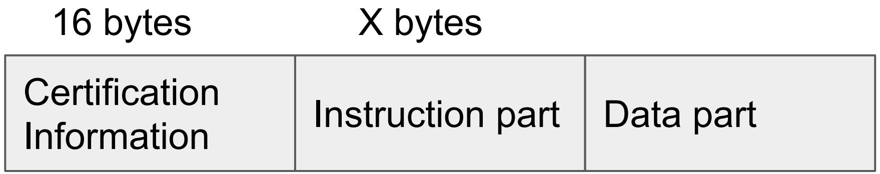
- Check client legitimation
- Fetch requested data & send back to client

Client:

- Process response from server
- Get requested data

Part 2: VMess Protocol

Client Request:



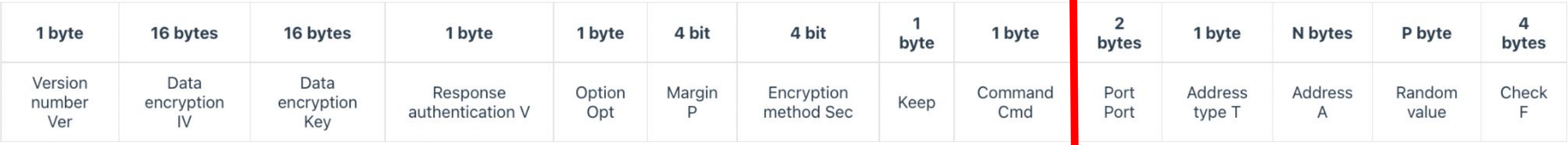
16 bytes	X bytes	
Certification Information	Instruction part	Data part

1. Certification Information: HMAC(H, K, M)
H: MD5 (recommend AEAD since 2020) (After 2022/01/01 MD5 deprecated)
K: Client User ID (16 bytes random number, works as a token)
M: UTC time (server will check time within 30 seconds)

Part 2: VMess Protocol

Certification Information	Instruction part	Data part

2. Instruction part: be encrypted by AES-128-CFB



1 byte	16 bytes	16 bytes	1 byte	1 byte	4 bit	4 bit	1 byte	1 byte	2 bytes	1 byte	N bytes	P byte	4 bytes
Version number Ver	Data encryption IV	Data encryption Key	Response authentication V	Option Opt	Margin P	Encryption method Sec	Keep	Command Cmd	Port Port	Address type T	Address A	Random value	Check F

Important elements:

- Encryption method for data part
- Margin P

Server first recv 38 bytes, process, and then recv N + 4 + P bytes, process

3. Data part: request content

Part 3: Weaknesses of V2Ray

(1) Unique TLS ClientHello Fingerprints

The previous version (<4.23.2) of V2Ray would send TLS ClientHello messages with very unique fingerprints, which allowed the censor to not only identify V2Ray client and server easily*, but block the traffic accurately as well.

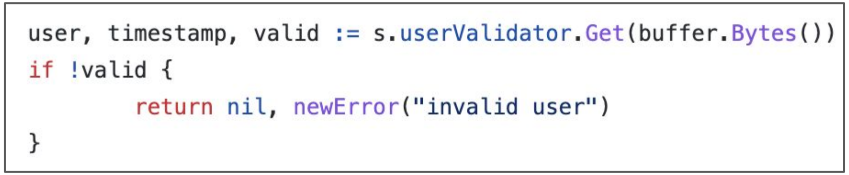
This vulnerability was caused by misuse of the Golang TLS library and a hardcoded cipher suite.

Mitigated in later versions by using the default settings.

*https://fr33land.net/2020/03/12/can-enable-tls-in-v2ray-help/ this blog proposed a neural network model to identify TLS traffic with 99.9% accuracy

Part 3: Weaknesses of V2Ray

v2ray-core/server.go

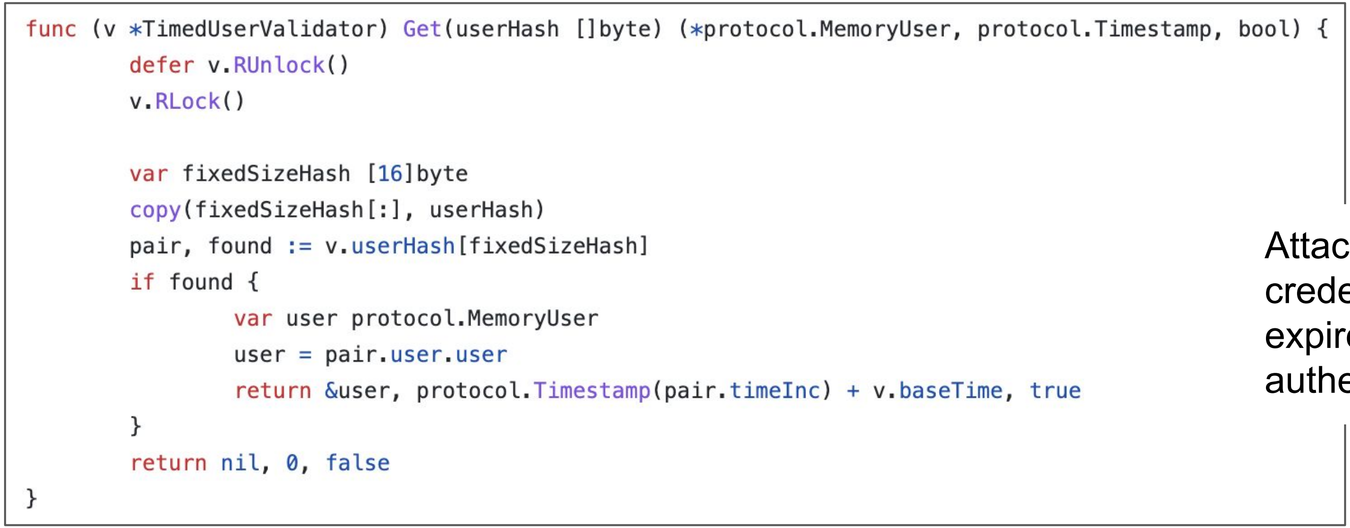


```
user, timestamp, valid := s.userValidator.Get(buffer.Bytes())
if !valid {
        return nil, newError("invalid user")
}
```

(2) Vulnerable to Replay Attacks

- Inappropriate authentication

v2ray-core/validator.go



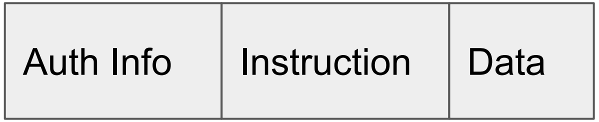
```
func (v *TimedUserValidator) Get(userHash []byte) (*protocol.MemoryUser, protocol.Timestamp, bool) {
        defer v.RUnlock()
        v.RLock()

        var fixedSizeHash [16]byte
        copy(fixedSizeHash[:], userHash)
        pair, found := v.userHash[fixedSizeHash]
        if found {
                var user protocol.MemoryUser
                user = pair.user.user
                return &user, protocol.Timestamp(pair.timeInc) + v.baseTime, true
        }
        return nil, 0, false
}
```

Attacker may reuse a valid credential before it gets expired to circumvent authentication!

Part 3: Weaknesses of V2Ray

(2) Vulnerable to Replay Attacks

- Because of the malleability of the stream cipher, attacker can measure the length of Data part `M` by forging legitimate requests with different `P` values. (This is similar to a "padding oracle".)
- Attacker may then guess the value of the 4-bit padding length Margin P in 16 tries. (details explained below)

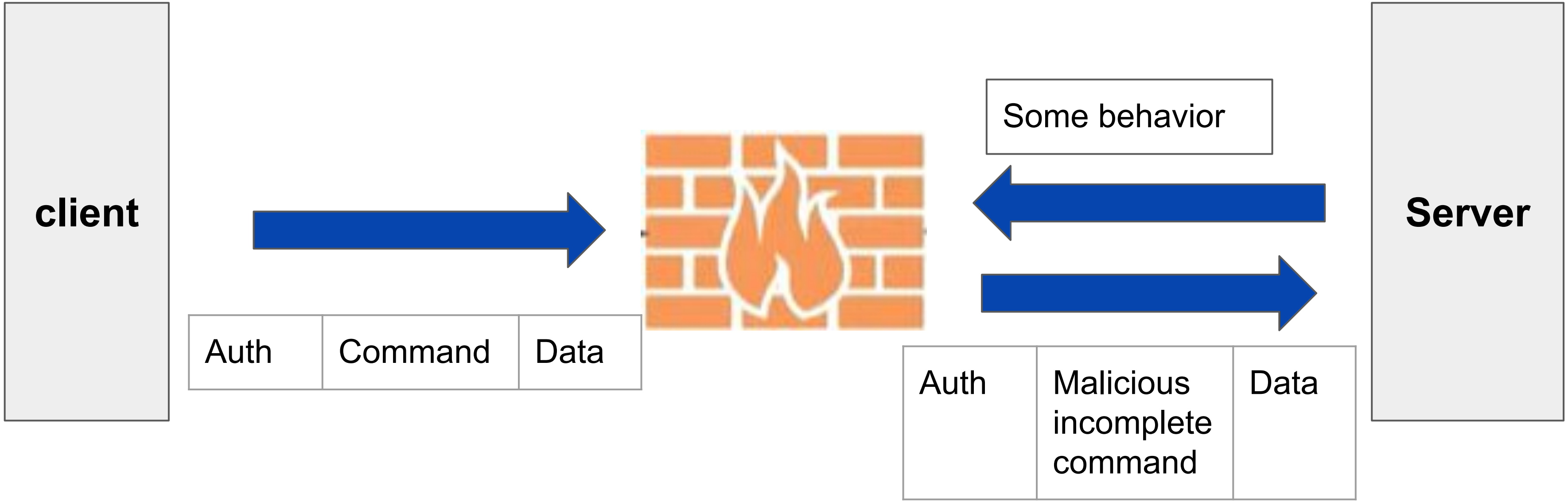


Auth Info

Instruction

Data

Part 4: Replay Attacks



client

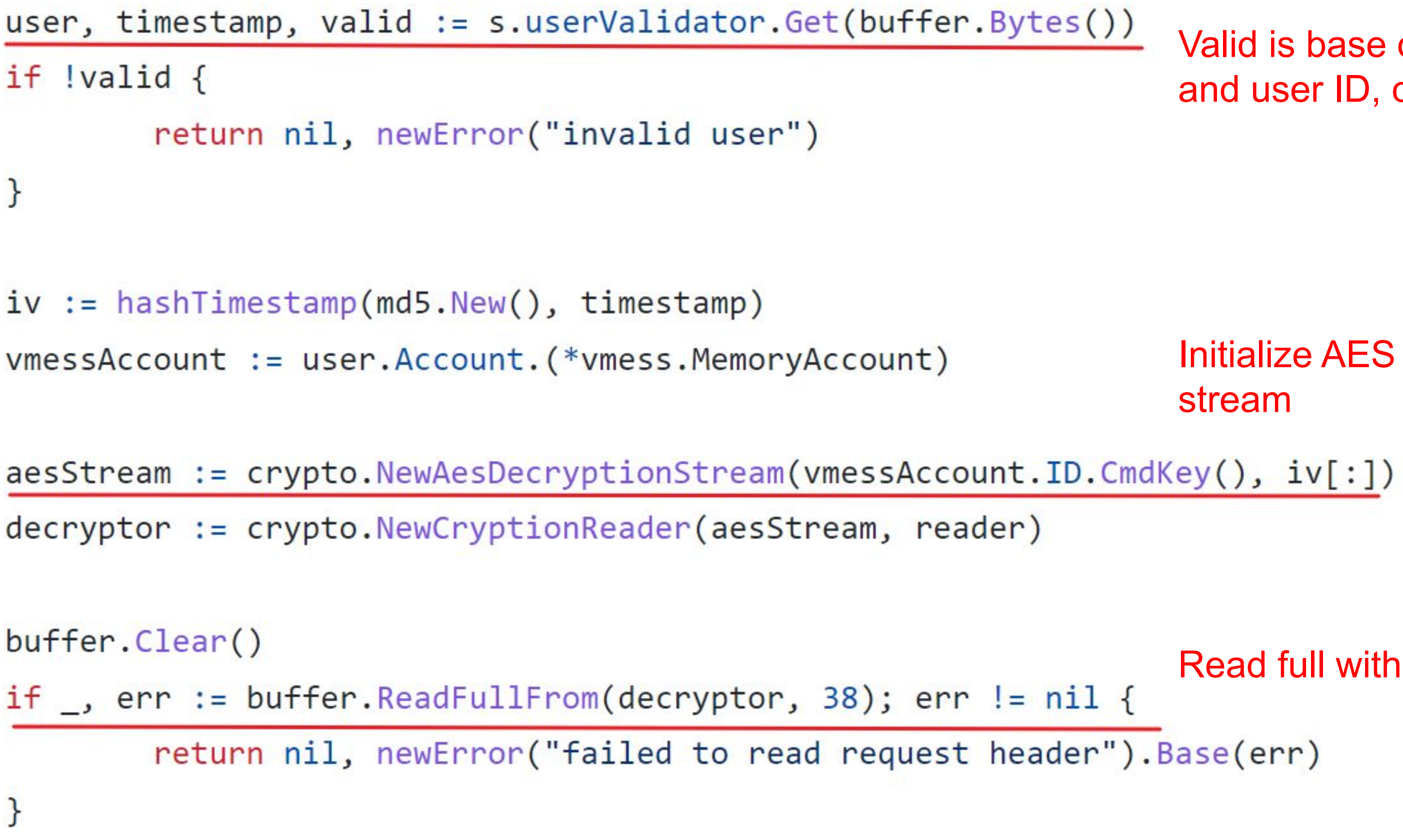
Auth | Command | Data

Some behavior

Auth | Malicious incomplete command | Data

Server

v2ray-core/server.go



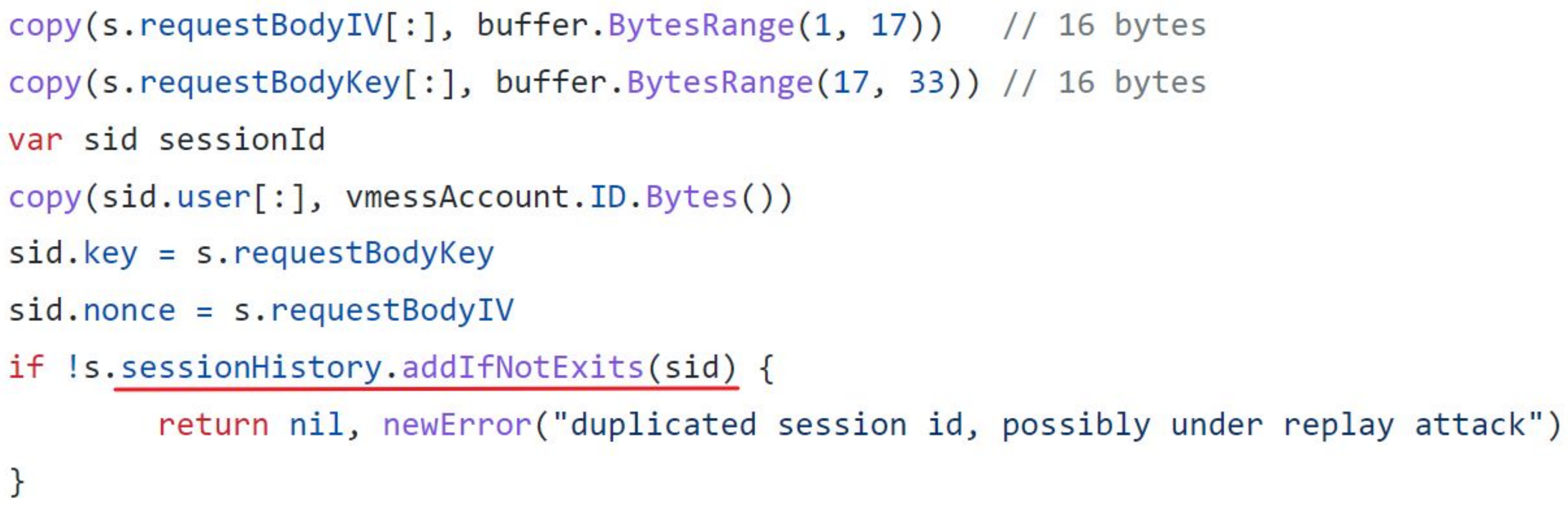
```
user, timestamp, valid := s.userValidator.Get(buffer.Bytes())
if !valid {
        return nil, newError("invalid user")
}

iv := hashTimestamp(md5.New(), timestamp)
vmessAccount := user.Account.(*vmess.MemoryAccount)

aesStream := crypto.NewAesDecryptionStream(vmessAccount.ID.CmdKey(), iv[:])
decryptor := crypto.NewCryptionReader(aesStream, reader)

buffer.Clear()
if _, err := buffer.ReadFullFrom(decryptor, 38); err != nil {
        return nil, newError("failed to read request header").Base(err)
}
```

Valid is base on timestamp and user ID, can be reused

Initialize AES decryption stream

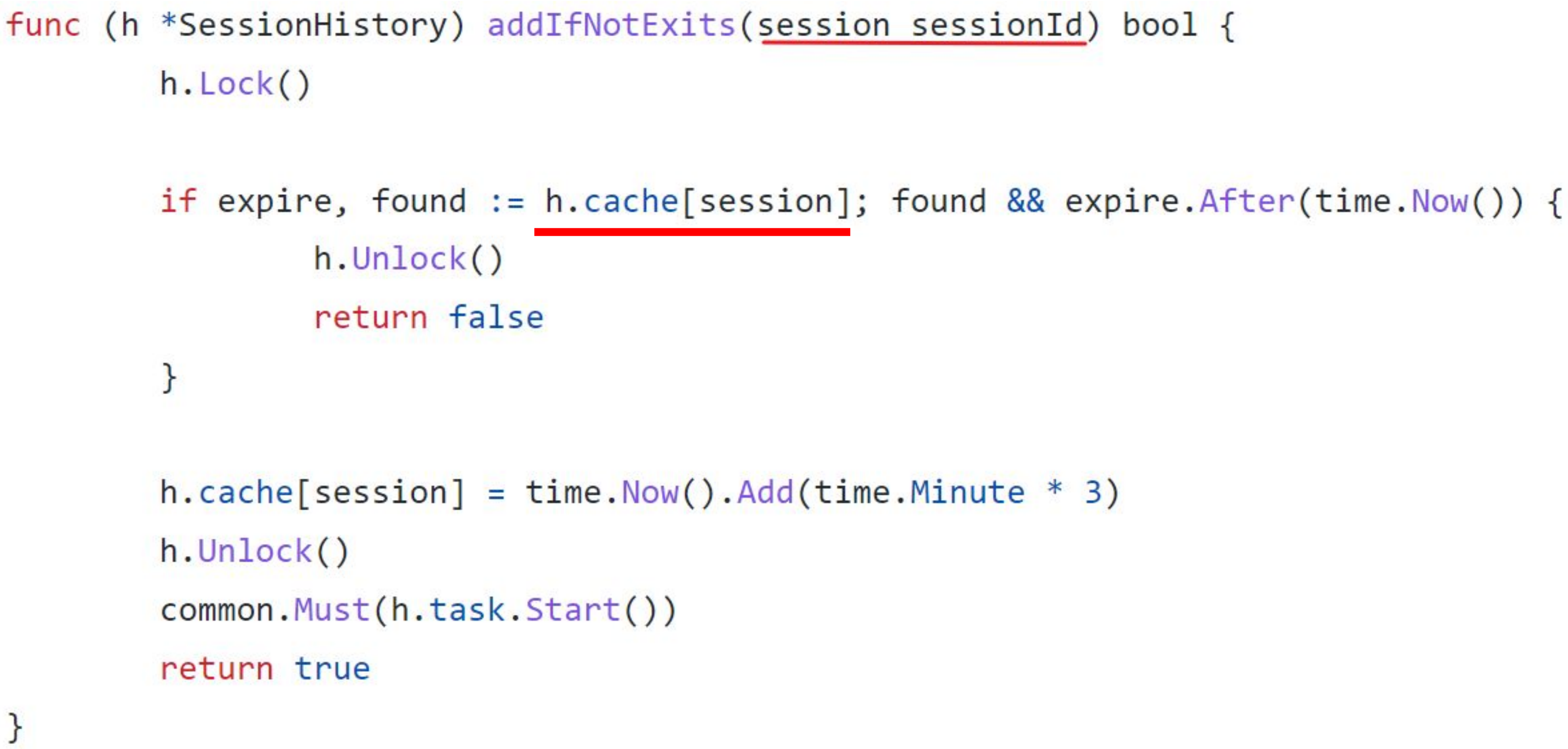
Read full without further validation

v2ray-core/server.go continued



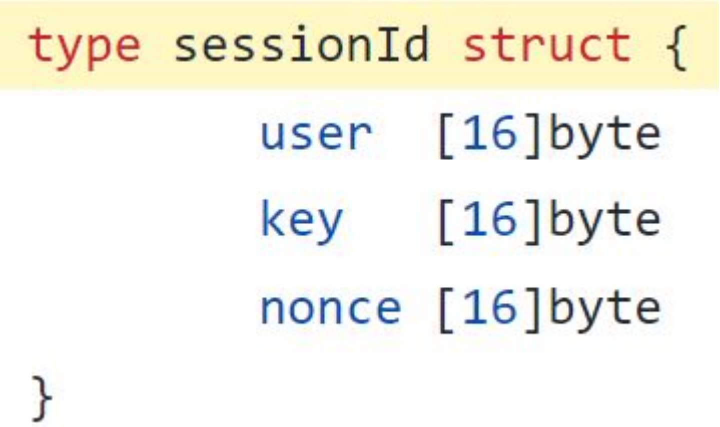
```
copy(s.requestBodyIV[:], buffer.BytesRange(1, 17))   // 16 bytes
copy(s.requestBodyKey[:], buffer.BytesRange(17, 33)) // 16 bytes
var sid sessionId
copy(sid.user[:], vmessAccount.ID.Bytes())
sid.key = s.requestBodyKey
sid.nonce = s.requestBodyIV
if !s.sessionHistory.addIfNotExits(sid) {
        return nil, newError("duplicated session id, possibly under replay attack")
}
```

Why sessionHistory does not work



```go
func (h *SessionHistory) addIfNotExits(session sessionId) bool {
        h.Lock()

        if expire, found := h.cache[session]; found && expire.After(time.Now()) {
                h.Unlock()
                return false
        }

        h.cache[session] = time.Now().Add(time.Minute * 3)
        h.Unlock()
        common.Must(h.task.Start())
        return true
}
```

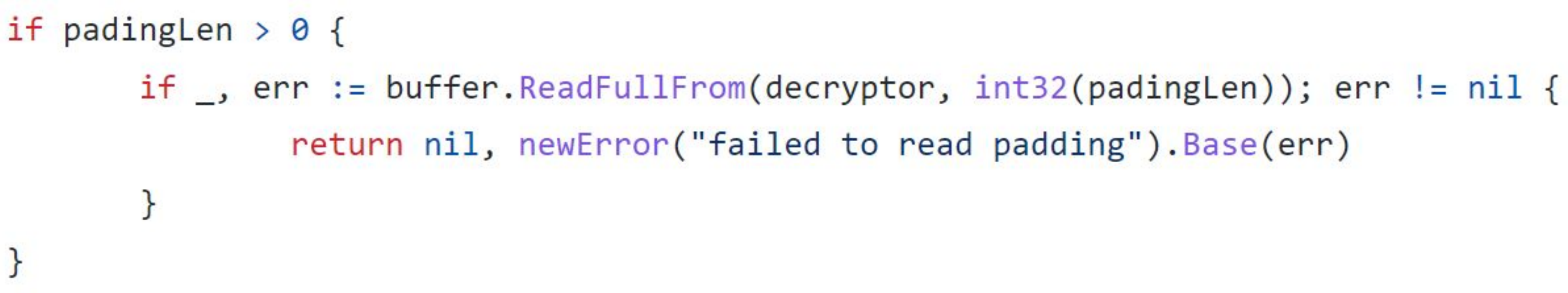
Session ID



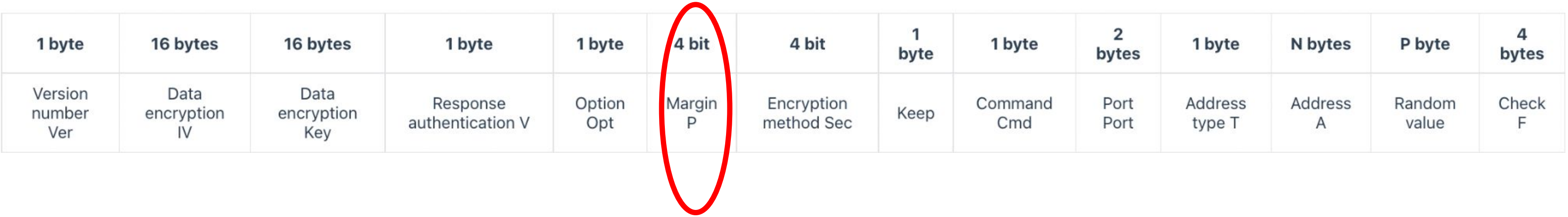
```
type sessionId struct {
        user  [16]byte
        key   [16]byte
        nonce [16]byte
}
```

If any field change, that’s a new sessionID!

v2ray-core/server.go continued



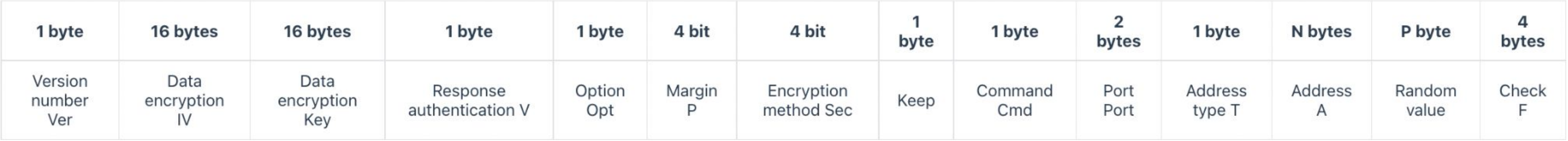
```
if padingLen > 0 {
        if _, err := buffer.ReadFullFrom(decryptor, int32(padingLen)); err != nil {
                return nil, newError("failed to read padding").Base(err)
        }
}
```



1 byte	**16 bytes**	**16 bytes**	**1 byte**	**1 byte**	**4 bit**	**4 bit**	**1 byte**	**1 byte**	**2 bytes**	**1 byte**	**N bytes**	**P byte**	**4 bytes**
Version number Ver	Data encryption IV	Data encryption Key	Response authentication V	Option Opt	Margin P	Encryption method Sec	Keep	Command Cmd	Port Port	Address type T	Address A	Random value	Check F

Replay attack

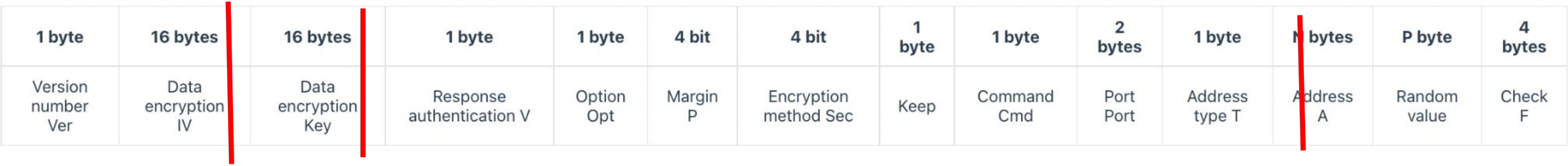
1. Send replay with the first 16+38 bytes of a valid connection, everytime change the last bit of Encryption key, and Margin P
2. Send M bytes of zero, wait until server closes the connection
3. Do step 1 and 2 for 16 times, record M values, if those values are X+0, X+1, X+2 … X+15, it is VMess connection



1 byte	16 bytes	16 bytes	1 byte	1 byte	4 bit	4 bit	1 byte	1 byte	2 bytes	1 byte	N bytes	P byte	4 bytes
Version number Ver	Data encryption IV	Data encryption Key	Response authentication V	Option Opt	Margin P	Encryption method Sec	Keep	Command Cmd	Port Port	Address type T	Address A	Random value	Check F

Why it works?

1. First 16 bytes are based on timestamp and userID, can be reused to pass validation
2. Changing the last bit of Encryption key can pass the check of sessionHistory
3. Because of instruction is AES-CFB encrypted, and this changed bit is in same block with Margin P, thus no error propagation

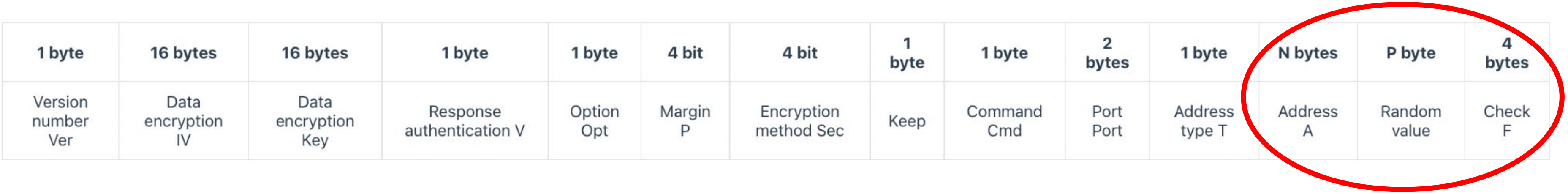


1 byte	16 bytes	16 bytes	1 byte	1 byte	4 bit	4 bit	1 byte	1 byte	2 bytes	1 byte	N bytes	P byte	4 bytes
Version number Ver	Data encryption IV	Data encryption Key	Response authentication V	Option Opt	Margin P	Encryption method Sec	Keep	Command Cmd	Port Port	Address type T	Address A	Random value	Check F

Why it works? continued

4. Server waited for address, padding, checksum, connection closed after found checksum is wrong

5. so M = address + padding + checksum, only padding length can change



1 byte	16 bytes	16 bytes	1 byte	1 byte	4 bit	4 bit	1 byte	1 byte	2 bytes	1 byte	N bytes	P byte	4 bytes
Version number Ver	Data encryption IV	Data encryption Key	Response authentication V	Option Opt	Margin P	Encryption method Sec	Keep	Command Cmd	Port Port	Address type T	Address A	Random value	Check F

Our Solutions

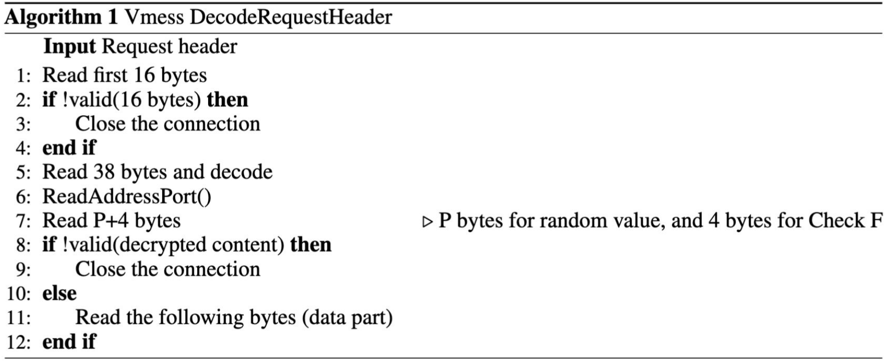
- Connection closes immediately after server finds checksum is wrong?

- What if the connection doesn’t close immediately?

- Randomly read several more bytes, and close the connection.

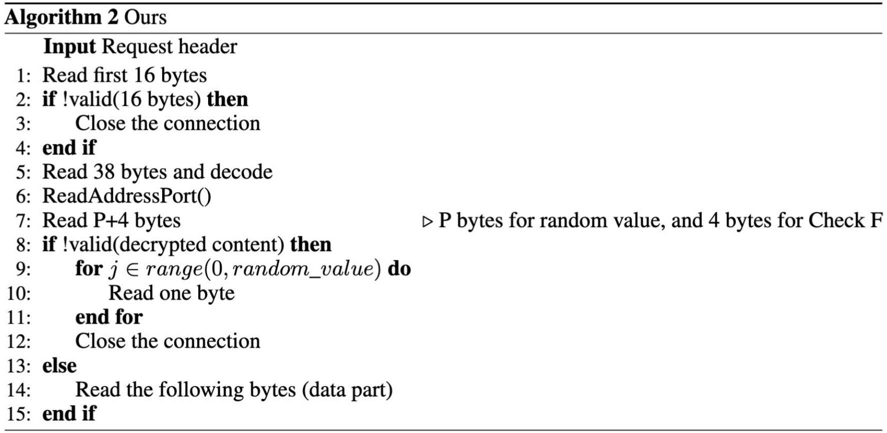
- 16 distinct values X+0, X+1, X+2 … X+15 -> it is VMess connection doesn’t hold true anymore.

Pseudocode



Algorithm 1 Vmess DecodeRequestHeader

```
Input Request header
Read first 16 bytes
if !valid(16 bytes) then
    Close the connection
end if
Read 38 bytes and decode
ReadAddressPort()
Read P+4 bytes                        ▷ P bytes for random value, and 4 bytes for Check F
if !valid(decrypted content) then
    Close the connection
else
    Read the following bytes (data part)
end if
```

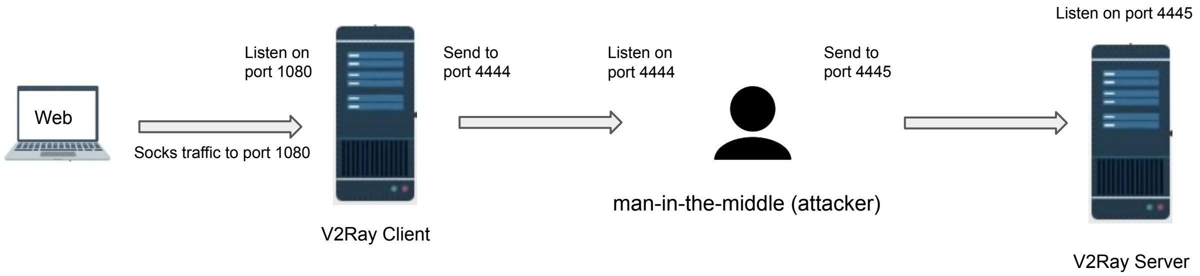


Algorithm 2 Ours

```
Input Request header
Read first 16 bytes
if !valid(16 bytes) then
    Close the connection
end if
Read 38 bytes and decode
ReadAddressPort()
Read P+4 bytes                        ▷ P bytes for random value, and 4 bytes for Check F
if !valid(decrypted content) then
    for $j \in range(0, random\_value)$ do
        Read one byte
    end for
    Close the connection
else
    Read the following bytes (data part)
end if
```

Experiment

- A simple version of V2Ray: https://github.com/jarvisgally/v2simple
- Replay attack: https://github.com/v2ray/v2ray-core/issues/2523#issue-628032465
- Our solution can successfully defend the replay attack


Web
Socks traffic to port 1080
Listen on port 1080
V2Ray Client
Send to port 4444
Listen on port 4444
man-in-the-middle (attacker)
Send to port 4445
Listen on port 4445
V2Ray Server