Traffic analysis of Secure Shell (SSH) - Trisul Network Analytics

dhivya ⋮ 15-19 minutes ⋮ 7/7/2017

Traffic analysis of Secure Shell (SSH)

Secure Shell (SSH) is a ubiquitous protocol used everywhere for logins, file transfers, and to execute remote commands. In this article, we are looking to use passive traffic analysis to detect various SSH events like login, keypress, and presence of SSH tunnels. Lets start with a question.

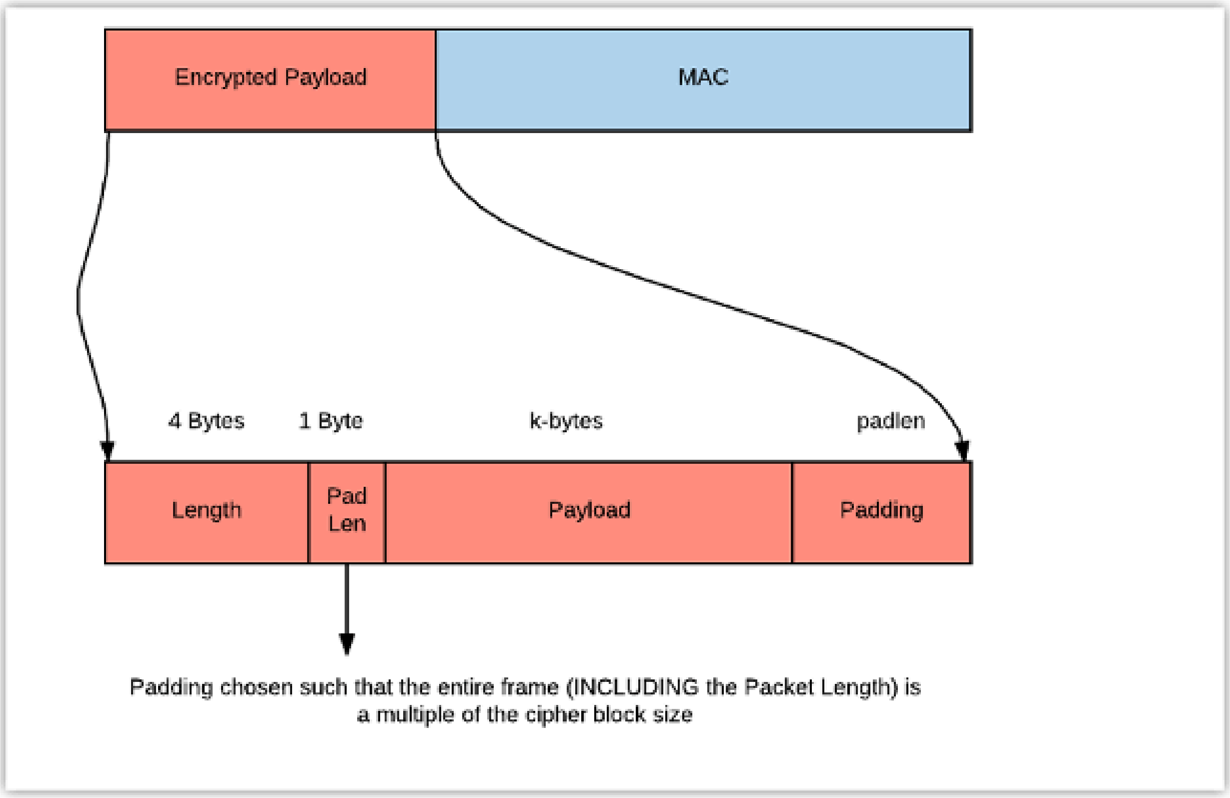
> What do you see on the wire when you press a key in a SSH terminal?

You could observe anywhere between 28 and 96 bytes of SSH payload depending on the type of secure channel negotiated. Lets dive a little deeper to see if we can get a accurate answer.

> **TLDR** Use traffic analysis to detect successful login , keystrokes, and Tunnels – reverse and forward. The approach is to use knowledge of the ciphers and MAC used in SSH and calculate the SSH message lengths on the wire. For login detection, we use the Terminal Capabilties Exchange , there are only a handful of terminal types so the message is predictable.

The basics : MACs and Ciphers

The SSH protocol offers both encryption and message integrity. Each packet is encrypted using a *Cipher* and authenticated using a *MAC*. If you capture packets using a tool like Wireshark, this is what a SSH record would look like. (without the TCP/IP headers)


Encrypted Payload
MAC
4 Bytes
1 Byte
k-bytes
padlen
Length
Pad Len
Payload
Padding
Padding chosen such that the entire frame (INCLUDING the Packet Length) is a multiple of the cipher block size

The general flow of the SSH protocol is

- The client and the server first exchange packets and agree on the MAC and Cipher algorithms. This is followed by a key exchange usually Diffie Hellman. This negotiation is unencrypted. In the real world, both directions use the same ciphers and MACs even though the SSH protocol itself does not mandate it.

Ciphers

Ciphers are used to encrypt your payload. Type `ssh -Q cipher` to get a list of supported ciphers by your client. You would see something like below. The comments are added by me

```
vivek@u14$ ssh -Q cipher

# old
3des-cbc blowfish-cbc cast128-cbc arcfour arcfour128 arcfour256
aes128-cbc aes192-cbc aes256-cbc rijndael-cbc@lysator.liu.se

# ctr mode AES - popular
aes128-ctr aes192-ctr aes256-ctr

# GCM mode AES - popular
aes128-gcm@openssh.com aes256-gcm@openssh.com  <-- gcm AEAD cipher

# New kid on the block
chacha20-poly1305@openssh.com
```

ETM vs non-ETM MACs — ETM stands for “Encrypt then MAC”. This represents a break from the older SSH which used “MAC then Encrypt”. Whether SSH negotiates an ETM or non-ETM MAC has a rather big implication for our traffic analysis.

The cryptographic doom principle and the SSH -etm MACs

The older non-ETM MACs like hmac-md5 first computed the MAC on the unencrypted SSH payload and *then* encrypted the message. This was MAC then Encrypt. I am not an expert on this and I can only guess why the initial SSH developers did this. The reason might have been to encrypt the packet length to thwart traffic analysis. Even today when you use hmac-sha2-256 you have to first decrypt the packet and then get the packet length. This means that you have to run a decryption operation on an unverified message.

Touching an unverified message is frowned upon by crypto experts

> “if you have to perform any cryptographic operation before verifying the MAC on a message you’ve received, it will somehow inevitably lead to doom.” — **Moxie Marlinspike**

OpenSSH now defaults to the *-etm ciphers. These first encrypt the payload then apply the MAC. The flip side is that now you need to transmit the packet length in the clear. Why? Because the receiver needs to check the MAC first before decryption. To do this you need to know the total length of the packet so you can find the MAC ! This exposes the protocol to traffic analysis but avoids the ‘moxie crypto doom principle’.

A single keystroke on the wire

Getting back to the traffic analysis of a single keystroke. We have the following so far

1. the handshake is unencrypted, so we can synchronize the Cipher/MAC state with the SSH session
2. the AEAD ciphers dont use a separate MAC
3. the *etm MACs transmit unencrypted packet lengths
4. the ciphers negotiated arent very important from a traffic analysis perspective as most of them are stream ciphers with a 16-byte block length except ChaCha20 which is 8 bytes

So lets jump in and see how this looks on the wire with a concrete example.

Say you hit a **Space Character ASCII 32 (Hex 20)** in a SSH session that has negotiated an **aes128-ctr** cipher. Per the SSH protocol RFC 4253 the encapsulation for carrying this single byte is the structure SSH_MSG_CHANNEL_DATA

```
 byte      SSH_MSG_CHANNEL_DATA
      uint32    recipient channel
      string    data
```

and the `string` datatype is a 4-byte length followed by the raw string data. The full encapsulation would then look like this

```
 byte      SSH_MSG_CHANNEL_DATA       93
      uint32    recipient channel          0
      uint32    stringlength               1
      byte[1]   content                   32  ( ASCII of space
character)

      Total = 1 + 4 + 4 + 1 = 10 bytes
```

Lets say we use HMAC-SHA1 (20 bytes) as the MAC algorithm. These 10 bytes when seen on the wire can be 40 or 52 bytes depending on whether the ETM mode is used or not.

sha1 Non ETM – 52 bytes per keystroke

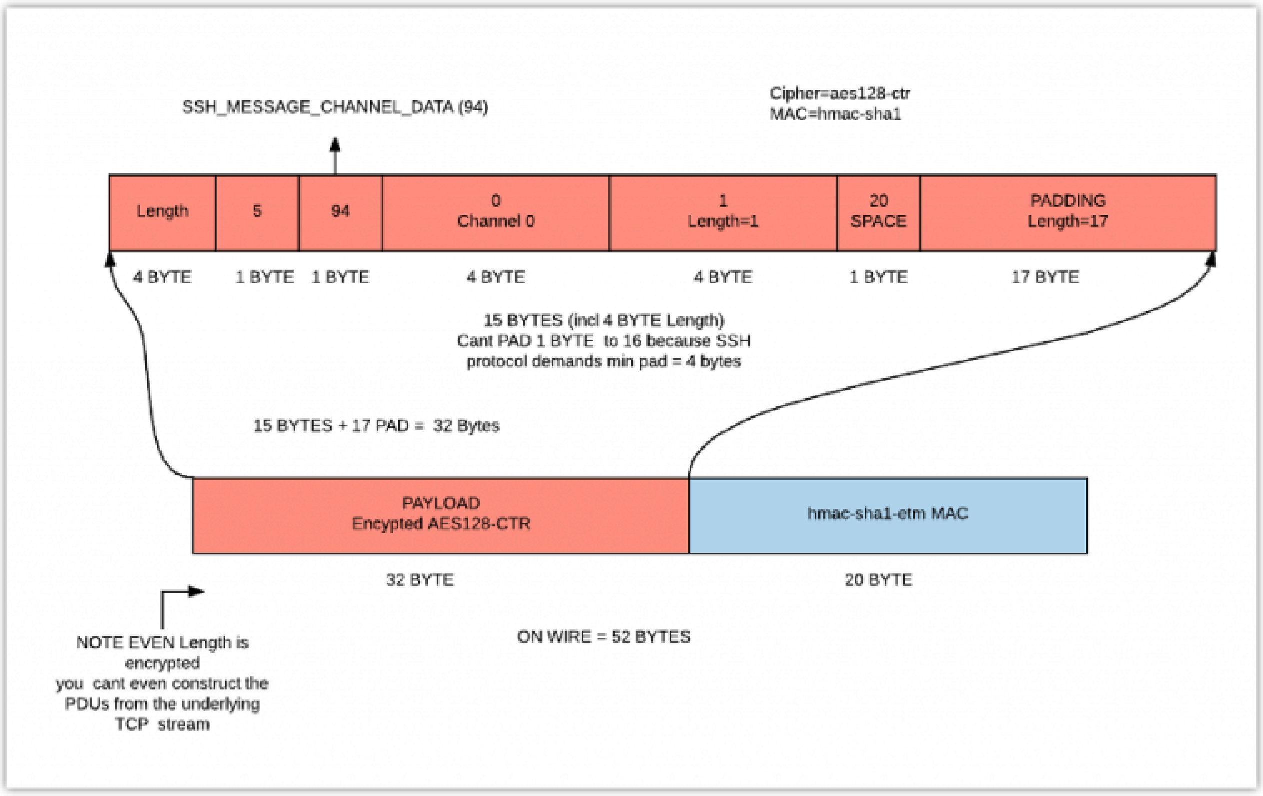
- When a non-ETM MAC Is used the packet length is also encrypted. Therefore the total payload to be encrypted is

```
uint32      length
        byte        padlen
        10 bytes    payload (SSH_MSG_CHANNEL_DATA for ASCII 20 )
        ??          random_padding
        Total without padding = 15 bytes
```

To get it to a 16byte boundary you need a padding of just 1 byte. But the "SSH Transport Layer RFC 4253 ":https://www.ietf.org/rfc/rfc4253.txt states

1. must pad to a multiple of cipher block size or 8 whichever is larger. Here the AES128-CTR block size is 16 bytes
2. must have a **minimum pad of 4 bytes**

So, we cant just pad 1 byte, we must now add 17 bytes of padding to bring it up to 32 bytes which is the next multiple of the cipher block size. The full packet looks like below.


SSH_MESSAGE_CHANNEL_DATA (94)
Cipher=aes128-ctr
MAC=hmac-sha1
Length
5
94
0
Channel 0
1
Length=1
20
SPACE
PADDING
Length=17
4 BYTE
1 BYTE
1 BYTE
4 BYTE
4 BYTE
1 BYTE
17 BYTE
15 BYTES (incl 4 BYTE Length)
Cant PAD 1 BYTE to 16 because SSH
protocol demands min pad = 4 bytes
15 BYTES + 17 PAD = 32 Bytes
PAYLOAD
Encypted AES128-CTR
hmac-sha1-etm MAC
32 BYTE
20 BYTE
ON WIRE = 52 BYTES
NOTE EVEN Length is encrypted
you cant even construct the PDUs from the underlying TCP stream

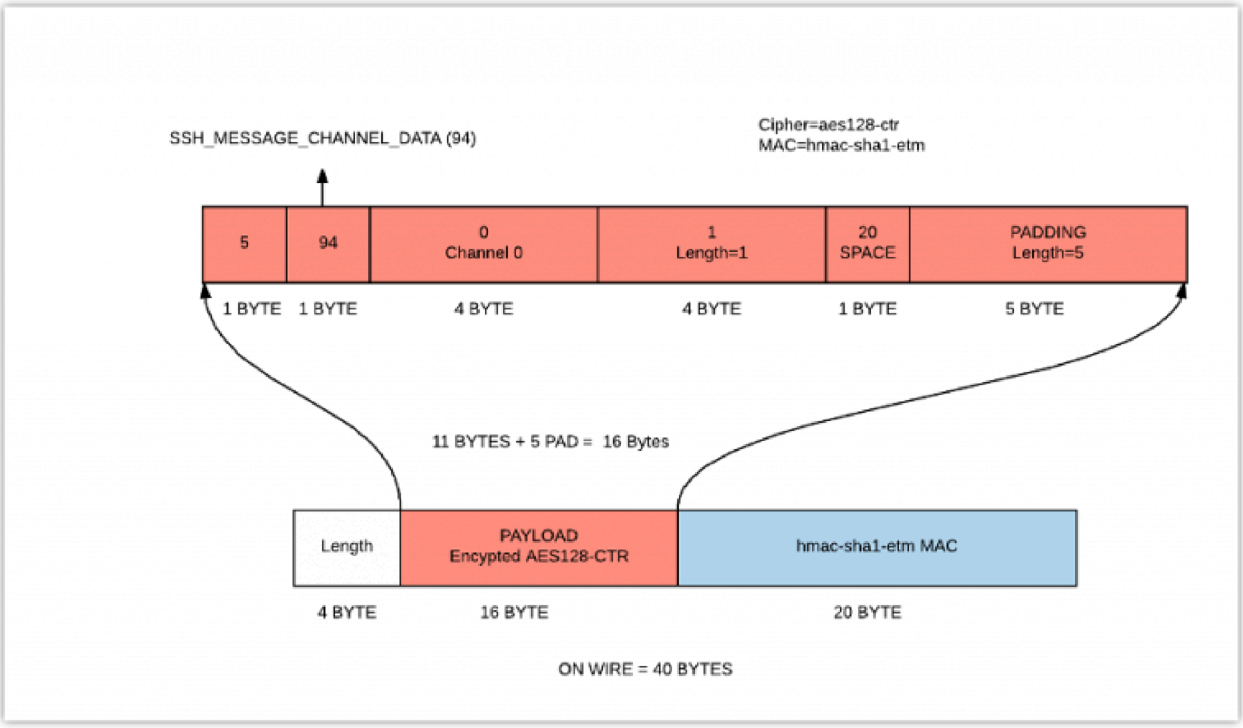
Again note that the packet length itself is encrypted. This thwarts passive reconstruction of SSH PDU (Messages) from a TCP byte stream. This is more resistant to traffic analysis but it breaks the Moxie Cryptographic Doom Principle by forcing the receiver to do a decrypt operation on unverified message. Thats where the ETM MACs come in.

sha1 ETM – 40 bytes per keystroke

By using an ETM MAC you not only get better security but the packet length drops from 52 to 40 bytes. Carrying on from the previous example. When you use an ETM MAC the packet length isnt encrypted. The message to be encrypted is

```
byte        padlen
        10 bytes    payload (SSH_MSG_CHANNEL_DATA for ASCII 20 )
        ??          random_padding
        Total without padding = 11 bytes
```

Now you need to add 5 bytes of padding to bring the total message up to the AES128-CTR cipher block length of 16. The good news here is that 5 bytes of padding is more than the mandated minimum of 4 bytes. The total length on the wire is 40 bytes as shown below



Here the packet length is unencypted. This allows traffic analysis tools like Trisul to reconstruct the SSH PDUs from a TCP byte stream.

chacha20-poly1305 – 36 with encrypted length

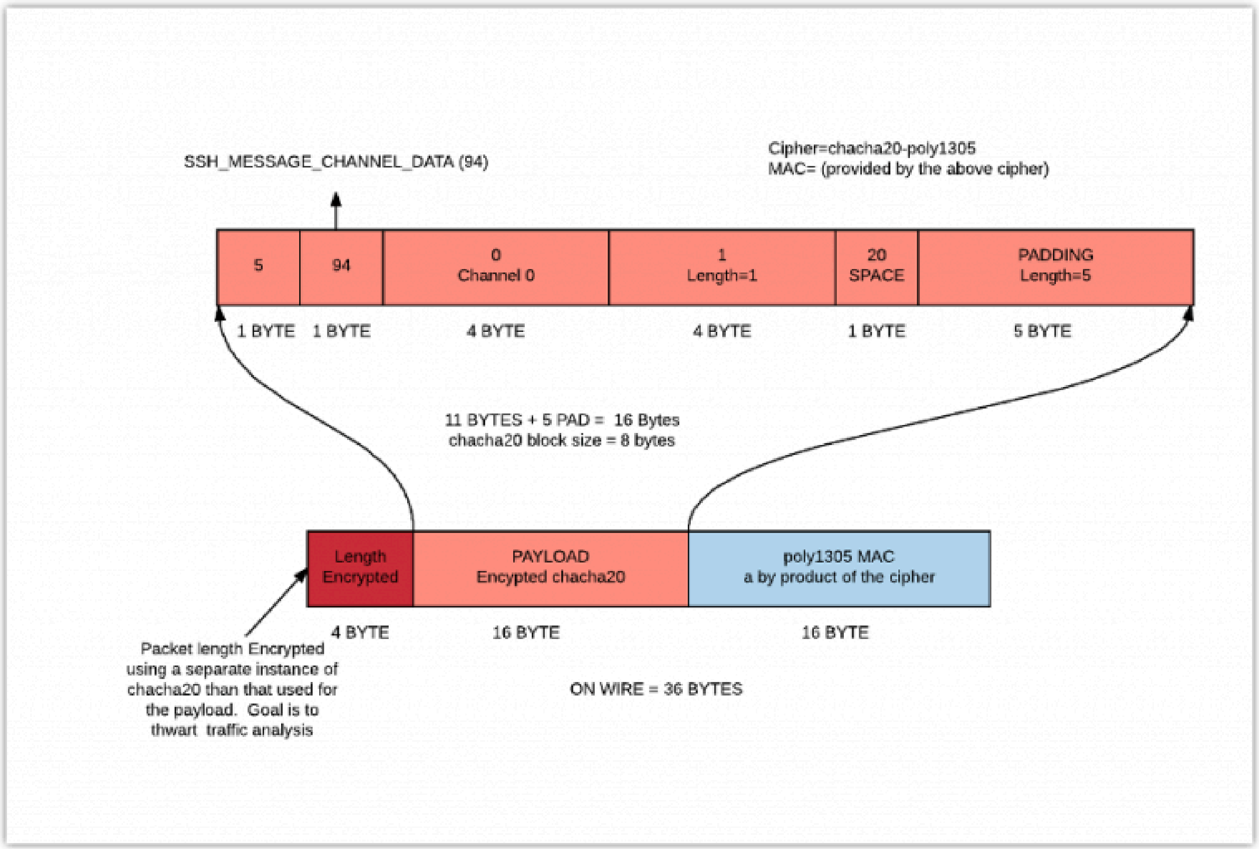
This particular cipher is currently zooming in popularity due to Chrome’s adoption. It is an AEAD cipher that is apparently computationally very efficient and it is as secure as AES. It is also the default cipher for SSH in latest versions of Ubuntu when I checked. ChaCha20-Poly1305 is worth a closer look because OpenSSH treats this cipher differently.

On the wire ChaCha20 is exactly like the ETM case, the packet length is not part of the encrypted message. The message is :

```
byte        padlen
        10 bytes    payload (SSH_MSG_CHANNEL_DATA for ASCII 32  )
        ??          random_padding
        Total without padding = 11 bytes
```

Now the block size of ChaCha20-Poly1305 is 8 bytes – so you need 5 bytes of padding to bring it up to 16 bytes. To that, add the 16 byte AEAD token created by Poly1305. You end up with total message size of 36 bytes. At this point the OpenSSH implementation adds a twist.

The chacha20 packet length field is also encrypted. They use a separate instance of ChaCha20 for this. I suppose this doesnt break the aforementioned “Moxie Doom” principle because the length isnt authenticated. It seems to me like the only reason OpenSSH does this is to thwart traffic analysis. The other AEAD ciphers AES-128-GCM do not encrypt the packet length. This how a single keystroke on a ChaCha20-Poly1305 SSH session looks like on the wire.


SSH_MESSAGE_CHANNEL_DATA (94)
Cipher=chacha20-poly1305
MAC= (provided by the above cipher)
5
94
0
Channel 0
1
Length=1
20
SPACE
PADDING
Length=5
1 BYTE
1 BYTE
4 BYTE
4 BYTE
1 BYTE
5 BYTE
11 BYTES + 5 PAD = 16 Bytes
chacha20 block size = 8 bytes
Length
Encrypted
PAYLOAD
Encypted chacha20
poly1305 MAC
a by product of the cipher
4 BYTE
16 BYTE
16 BYTE
Packet length Encrypted using a separate instance of chacha20 than that used for the payload. Goal is to thwart traffic analysis
ON WIRE = 36 BYTES

Detect a successful login

A naive way to detect a successful login is to alert when a total traffic exchanged on a SSH flow is greater than a threshold value. We currently use that with Trisul and the results are hit and miss. Few factors that can confuse us:

1. Use of login banners with variable length strings
2. Pub key authentication involve exchanging more data

Turns out there is a much better way to detect login. After a successful password or key authentication, the client does two things in succession

1. sends a SSH_MSG_CHANNEL_REQUEST —> requests a pseudo terminal
2. sends a SSH_MSG_CHANNEL_REQUEST —> requests a shell

It turns out there arent that many different types of terminals so these two messages are mostly of fixed length.

Message requesting a pseudo-terminal- per RFC 4254 Secure Shell Connection the message is

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    “pty-req”
boolean   want_reply
string    TERM environment variable value (e.g., vt100)
uint32    terminal width, characters (e.g., 80)
uint32    terminal height, rows (e.g., 24)
uint32    terminal width, pixels (e.g., 640)
```

```
uint32    terminal height, pixels (e.g., 480)
string    encoded terminal modes
```

The only variable part of this message is the “encoded terminal modes”. There arent that many types of terminals so this part is mostly fixed. **This message is 312 bytes for all linux based terminals** we tested and 288 bytes for PUTTY versions. The difference is due to missing terminal modes in putty.

The second channel request message for “shell” is

```
yte      SSH_MSG_CHANNEL_REQUEST
uint32    0
uint32    5  (string length)
5 bytes   “shell”
1         want reply
```

This is 16 bytes when login sessions ask for `shell`

For *etm MACs, the packet lengths are directly transmitted, since 312 bytes are padded to 320 (multiple of 16) you look for 320 bytes.

```
# for *-etm - you can read the packet length directly
        # so that is really easy
        termcap = 320
```

For chacha20 and the non-ETM MACs the packet lengths are encrypted. So you have to look for TCP reassembled chunks that match the TERMCAP and SHELL since they are typicaly sent together.

```

        # for chacha20-poly1305 312 bytes and the 20 bytes are sent
together
        4 + 312 + 16  = 332
        4 + 24 + 16   = 44
                Total = 376
```

For chacha20-poly1305 you look for TCP chunk updates of 376 bytes.

In a similar manner you can construct a table for each of the ciphers and MACs. See ssh_dissect.lua on our Github page.

Detecting keystroke after login

As we saw a single keystroke results in 10 byte payload, which can be easily translated to on-wire lengths

1. for *etm MACs and GCM ciphers – the packet length is directly sent.
2. for chacha20 you look for TCP window updates of 36 bytes

3. you also need to add in a timing check. Since *Pseudo-TTYs* echo the characters you type. If you detect a client keypress and immediately within (N seconds) a server echo, you can declare a keypress event.

Detecting SSH Tunnels

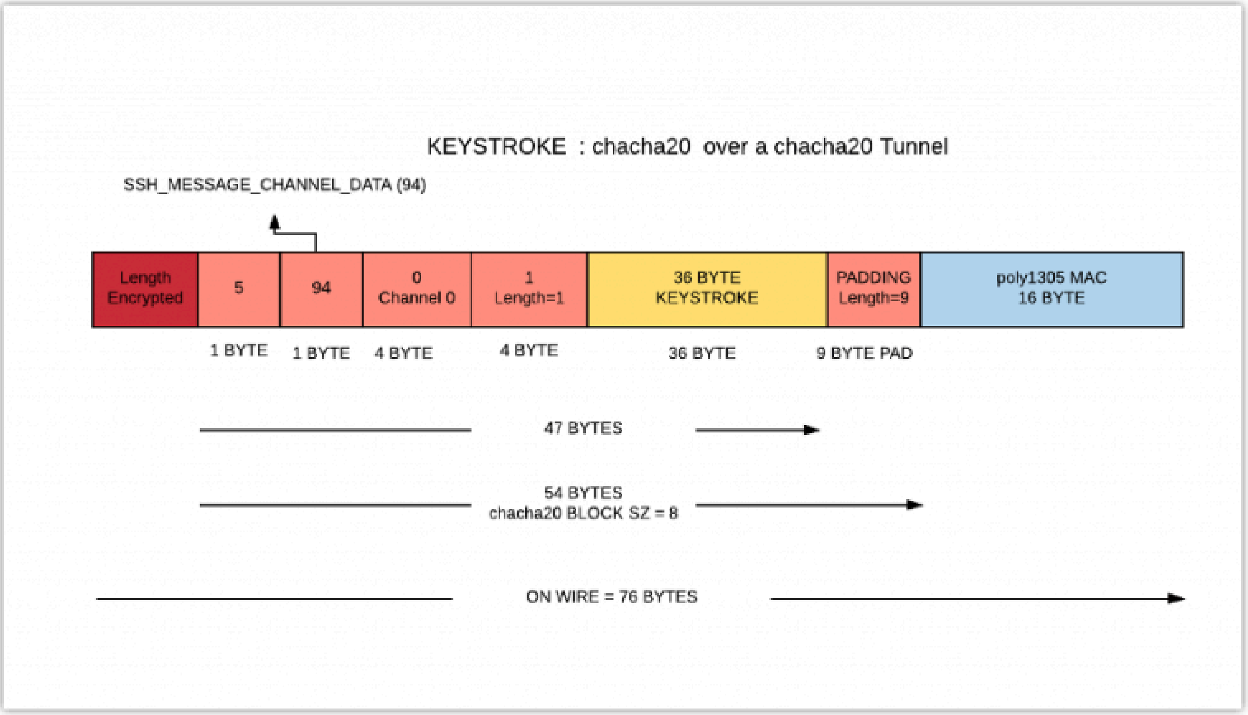
This was the real motivation for me looking at this protocol in detail. SSH Tunnels are completely opaque to systems like Trisul/Bro/Suricata which look at packets. If a hostile element manages to establish an autossh reverse tunnel out of your network, you are really pWnd. He can come and go deep into your network at will, no firewall or IDS, IPS will detect that.

We had an earlier implementation mentioned in this blog Detecting Reverse SSH Tunnels that used a range of packet lengths to detect a tunnel. It was a reasonable approach but a lot would have slipped through because we didnt explictly track the MACs and Ciphers.

With SSH Tunnels there are two sessions.

1. **Parent session** – this has its own MAC/Cipher negotiated
2. **Tunnel sessions** – has a separate MAC/Cipher negotiated – ALL SSH messages on this tunnel are carried inside a “Channel” on the parent tunnel.
3. **Channels** — The parent SSH tunnel itself can be a live interactive session. SSH distinguishes tunneled traffic from through traffic using the *Channel Number* The Parent session is Channel 0 and the tunnels are Channel 1, 2, 3 etc.
4. **Reverse tunnel** — There is no difference on the wire between a Forward SSH Tunnel (`ssh -L ..`)or a Reverse SSH Tunnel (`ssh -R ..`)

Example of a single keystroke inside a SSH Tunnel. We have seen that a single keystroke in a SSH session using chacha20-poly1305 is 36 bytes. Now imagine that this is now carried inside a SSH tunnel that is also using chacha20-poly1305. How does one calculate the packet length? A single keystroke on a chacha20 SSH session that is run over a chacha20 tunnel is **76 bytes**. The computation is quite simple if you followed the previous examples. Here is how that looks. Notice that ALL the traffic from the tunnel is carried inside a Channel 1 wrapped by the MSG_CHANNEL_DATA.



Once we got this far, we apply the same techniques to calculate a lookup table. See ssh_dissect.lua

Programming detection into your tool

Given the unmistakable move towards total encryption, traffic analysis techniques are going to become more important for “network security monitoring” platforms. Recently Damien Miller , an OpenSSH committer blogged about how thwarting traffic analysis was a high priority in his article ChaCha20 and Poly1305 in OpenSSH

Programming this in your tool of choice would include :

- in the clear part of the SSH handshake process : synchonize the cipher/mac used
- **Successful login** : When the packet lengths for Terminal Capability and the Shell request are seen
- **Keystroke after login** : When the packet length for single key press and response are seen
- **SSH Tunnels** : Packet lengths for tunnel key press are seen
- The reassembly offered by the platform must be **real time and unbuffered**. This means that as soon as the TCP Chunks arrive and are reassembled they must be made available to the traffic analysis script.
- You can turn off reassembly after the initial detection. To detect tunnels you only need the TCP Chunk sizes not a full blown reassembly.

How can SSH developers really thwart traffic analysis? Here are a couple of ideas from the top of my head

1. Implement the random padding requirement in RFC 4253 “Secure Shell Transport Protocol”. Today the OpenSSH suite pads up to the cipher block boundary they can extend it to pad to a random boundary . In my view, this has the drawback that the random padding has to be a multiple of block size so there are only so many discrete choices. Adding too much padding – say 512 bytes – for a single keystroke is just way too much overhead. On the other hand if you only pad to a few known values, we can easily account for those in the detection.
2. Use a technique like the TLS GREASE extensions to throw off traffic analysis by inserting bogus Terminal Capabilities or NOOP bogus keystrokes. This doesnt make much sense because unlike the TLS handshakes that GREASE focuses on, the SSH messages are encrypted. Bogus keystrokes could be hard to handle though.
3. Throw off TCP by buffering. This is just wicked. When TCP increments do not corresponding to the message lengths pushed it would be impossible to know when one message ends and another starts. They dont impact the **-etm* MACs but for *ChaCha20-Poly1305 and non-ETM* this would be a dead end. We will see if the SSH developers are willing to go this far.

You can find the detection scripts we created for Trisul on our Github Repo trisul-scripts

Trisul 6.0 ! Ready to go packages for Ubuntu and CentOS.