What?

Imagine an Internet where every packet is cryptographically protected from source to destination against espionage and forgery, getting an IP address is as simple as generating a cryptographic key, core routers move data without a single memory look up, and denial of service is a term read about in history books. Finally, becoming an ISP is no longer confined to the mighty telecoms, anyone can do it by running some wires or turning on a wireless device.

This is the vision of cjdns.

Why?

The Internet is built on protocols which largely date back to the late 80's or earlier. At a time when it was a network of anarchistic academics and scholars showing the ITU that open standards matter, it was absolutely enough. Over time the network has gotten bigger and the users have found new needs.

In the age when packet inspection is universal and security breaches are commonplace, cryptographic integrity and confidentiality are becoming more of a requirement. The US government recognized this requirement and has been helping through IPSEC and DNSSEC efforts.

Another issue is how are we going to route packets in a world where the global routing table is simply too large for any one router to hold it all? Despite the heroic efforts of core network engineers, the growth of the global routing table seems an unstoppable march. Cisco router company has proposed a plan called Locater/Identifier Separation Protocol, or LISP which aims to solve this by re-aggregating the routing table without forcing people to change their precious IP addresses. A different view of this problem is IP address allocation, currently it is done by a central organization which assigns IP addresses in such a way as to make the routing table as small as possible. Unfortunately this creates a bar of entry to the ISP sphere because aspiring network operators must register with the central organization and apply for an allocation of IP addresses while demonstrating that they will not be wasted. It is always easier to show that you need IP addresses if you already have a network.

Denial of service, an attempt to prevent legitimate users from accessing a service[1], is likewise a new problem in the expanding network. To my knowledge, there is no general purpose solution to denial of service attacks. Solutions to packet flood based denial of service often revolve around hosting a service on many computers so that they can handle an enormous amount of traffic.

Finally, the existing protocols are difficult to use, we cannot reasonably assign blame to anyone for this, many of these protocols are over thirty years old and demonstrate a level of craftsmanship which I can only hope to one day achieve.

However, thirty years takes its toll on the best of us and as the Internet grew and became more complex, the administration interface of the typical router has grown a thicket of knobs, buttons and switches to match the proliferation of use cases and failure modes. As a result, network operation has become a science where students receive degrees and certificates for knowing the meanings of the plethora of knobs and switches, it has also become, like the tuning of the race car, an art, passed from master to apprentice and shared on mailing lists. Suffice to say, the bar of entry into the ISP realm is too high. Users, particularly in the ad-hoc wireless network arena, have observed the high bar of entry into traditional routing and have developed a menu of alternative, self-configuring protocols such as OSLR, HSLS, and BATMAN.

So the problems are already solved?

Not every problem listed has an existing solution and of the ones which do, many of the solutions are based on incompatible technology. For example: OSLR was not designed to operate with IPSEC and LISP. Even where the solutions exist and are ready for deployment, they still require mass technology adoption and they don't offer existing ISPs significant immediate gains.

The mismatch is rather absurd. On the one hand there are scholars, engineers, hardware and software designers with combined over 1000 years of experience. There are mathematical formulae, proofs, papers and specification documents; written, circulated, peer reviewed and written again. On the other hand you have a single volunteer developer, a clean slate, and an attitude that nothing is impossible. How can this be anything short of lunacy?

In revolutionary times, the old book only weighs you down.

cjdns is built on the idea that the ISPs and hosting providers which exist now will never upgrade, not to LISP, not to DNSSEC, not even to IPv6 in any meaningful way. Building new systems to be compatible with old systems is catering to the audience you will never have. Asking existing ISPs to upgrade for the common good is asking them to take a risk with no immediate benefit. cjdns is about throwing out the book and redefining the specifications in a way that will be fast, secure, and most importantly, *easy* for the next generation of ISPs to deploy and use.

What is this denial of service?

Usage of a service can be interrupted by sending a flood of unwanted packets to a host from a large number of infected "zombie" machines. This, known as DDoS, is a problem which worsens every year as the upload speed of all infected nodes on the Internet grows in proportion to the download speed of any given link. Being infected with a virus and participating in DDoS, though not a picnic, is not an emergency for the owner of the infected machine. Nor is it an emergency for their ISP. DDoS is always "their problem"... Until it strikes in your network. Sadly, a common response from a datacenter is to stop carrying the controversial content, making DDoS an effective censorship tool and encouraging the practice.

Another form of denial of service which is even more insidious is intimidation by threat of faux court action. This form of denial of service is especially effective since most people get their IP addresses from their ISP, when their ISP is threatened, they need to make a judgment call as to the validity of the claim and they often act as judge and jury, disconnecting a customer in order to avoid conflict. Those who have their own IP addresses assigned to them, are able to essentially be their own ISP and to peer with a multitude of other ISPs making it very difficult to threaten anyone but them.

What is the routing table and why does it keep getting bigger?

A more technical issue with the Internet, and one of which many people are unaware, is address space deaggregation. Every computer connected to the Internet needs an address, a number which uniquely identifies it and which is attached to every piece of data which is to be sent to that computer. At every stop along its path through the Internet, a packet (unit of data) has its address field examined by a router so it can decide which wire that packet should be sent down. Routers have an easier time if addresses are in big blocks so that a router can look quickly at the first numbers in the address and know, for example, that it is destined for somewhere in China, not exact but enough to know which wire to send it though. People naturally want as many addresses as they can possibly get and they want them in the smallest blocks possible, this is so they can then control (or buy and sell) these small blocks independently. The smaller the blocks of addresses which are announced, the larger the routing tables become and the more work the Internet's core routers must do in order to send a packet in the right direction. There have been attempts to aggregate addresses back in to groups but nonetheless, the number of small announcements in the global routing table has grown every year.

To address the growing routing tables, Cisco has proposed a new protocol called Locator/Identifier Separation Protocol or LISP. The idea of LISP is to separate the addresses which people use from the addresses which routers use, like a lower level version of DNS. LISP allows the edge ISPs and users to see the Internet as they want it, deaggregated into small pieces for political reasons, and the routers to see the Internet as they want it, centralized hierarchical pyramid of addresses emanating from some arbitrary center point. This design works well in existing routers since they are designed to get packets with a universally unique address and look that address up in a table. This is advertised as a design feature but LISP is limited in its vision, if one must look up the "real location" of a server before forwarding a packet, why not simply look up the fastest path?

I don't care, it's their problem.

Each of these problems is a tragedy of the commons problem. The users of virus infected computers are incentivized to save money rather than purchasing a product or service to rid their computer of the infection. While there are solutions such as egress filtering which decrease the problem, ISPs are incentivized to implement as little security as possible because they are not directly affected. Denial of service, whether by packet flooding or by faux legal action, benefits the attacker who is able to hide the truth or victimize service providers as well as organizations who make it their business to provide DoS related services. The victim who all too often publishes information for no other reason than satisfaction of telling the truth, is the only party harmed by this type of attack, and he is the least able to prevent it. Address space deaggregation benefits the edge ISPs who gain more flexibility in how their network is organized at the cost of the core ISPs whose only defense is the "we will not route that" nuclear option which would no doubt bring about a revolt from the edge ISPs.

Each of these problems hurts everyone, DDoS forces ISPs to over prevision their lines, denial of service through faux legal action increases the cost of running a community website or ISP since each accusation must be reviewed and its validity assessed, and address deaggregation means everyone must pay more to have their packets routed through increasingly high power routers, and difficulty of operating a router and getting a block of IP addresses hurts competition in the ISP sphere, thus increasing prices and impeding progress.

How?

cjdns is made of three major components which are woven together. There is a switch, a router, and a CryptoAuth module. With total disregard for the OSI layers, each module is inherently dependent on both of the others. The router cannot function without routing in a small world which is made possible by the switch, the switch is blind and dumb without the router to command it, and without the router and switch, the CryptoAuth has nothing to protect.

The Switch

He doesn't think about his actions; they flow from the core of his being.

The switch design is unlike an IP or Ethernet router in that it need not have knowledge of the globally unique endpoints in the network. Like ATM switching, the packet header is altered at every hop and the reverse path can be derived at the end point or at any point along the path but unlike ATM, the switch does not need to store active connections and there is no connection setup.

Definitions

- Interface: A point-to-point link to another cjdns switch. This may be emulated by Ethernet frames, UDP packets or other means.

- Self Interface: A special Interface in each switch, packets sent for this interface are intended for the node which this switch is a part of. Upon reaching the ultimate hop in its path, a packet is sent through the Self Interface so it can be handled by the next layer in the node.
- Director: A binary codeword of arbitrary size which when received by the switch will direct it to send the packet down a given Interface.
- Route Label: An ordered set of Directors that describe a path through the network.
- Encoding Scheme: The method by which a switch converts one of its internal Interface ID (EG: array index) to a Director and converts a Director back to its internal representation. Encoding schemes may be either fixed width or variable width but in the case of variable width, the width must be self-describing as the Directors are concatenated in the Route Label without any kind of boundary markers.
- Encoding Form: A single representation form for encoding of a Director. For a given Encoding Form, there is only one possible way to represent a Director for a given Interface. A variable width Encoding Scheme will have multiple Encoding Forms while a fixed width Encoding Scheme will have only one.
- Director Prefix: For switches which implement variable width encoding, the least significant bits of the Director is called the Director Prefix and is used to determine the width of the Director. **NOTE** Because the Route Label is read from least significant bit to most significant bit, the Director Prefix is actually the bits furthest to the *right* of the Director.

Operation

When a packet comes in to the switch, the switch uses its Encoding Scheme to read the least significant bits of the Route Label in order to determine the Director and thus the Interface to send the packet down. The Route Label is shifted to the right by the number of bits in the Director, effectively *removing* the Director and exposing the Director belonging to the next switch in the path. Before sending the packet, the switch uses its Encoding Scheme to craft a Director representing the Interface which the packet came *from*, does a bitwise reversal of this Director and places it in the empty space at the left of the Route Label which was exposed by the previous bit shift. In this way, the switches build a mirror image of the return Label allowing the endpoint, or any hop along the path, to derive the return path by simple bitwise reversal without any knowledge of the Encoding Schemes used by other nodes.

Example

Supposing Alice wanted to send a packet to Fred via Bob, Charlie, Dave and Elinor, she would send a message to her switch with the first Director instructing her switch to send down its interface to Bob, the second Director instructing Bob's switch to send to Charlie and so on.

NOTE: Spaces between bits are for illustration only, Route Labels have no boundary markers between Directors and switches cannot know how many bits a Director uses unless that Director is their own.

Alice's original Route Label, before entering her switch:

```
0000000000000000000000000 0001 101011 011010 100101101 10111 0100011
^^^-- unused space --^^^^                                    ^^^^^^^-- Alice's Director Interface(Alice->Bo
```

Route Label when it reaches Bob:

```
1000000 0000000000000000000000000 0001 101011 011010 100101101 10111
^^^^^^^                                                         ^^^^^-- Bob's Director for Interface(Bob->Ch
      ^-- Alice's Director for her Self Interface (reversed)
```

Route Label when it reaches Charlie:

```
11001 1000000 0000000000000000000000000 0001 101011 011010 100101101
^^^^^ ^^^^^^^                                              ^^^^^^^^^-- Charlie's Director for Interface(Cha
  ^^^       ^-- Alice's Director for her Self Interface (reversed)
    ^-- Bob's Director for Interface(Bob->Alice), bit-reversed.
```

Route Label when it reaches Dave:

```
110110011 11001 1000000 0000000000000000000000000 0001 101011 011010
^^^^^^^^^ ^^^^^ ^^^^^^^                                       ^^^^^^-- Dave's Director for Interface(Dave->
   ^^^^^^   ^^^       ^-- Alice's Director for her Self Interface (reversed)
      ^^^     ^-- Bob's Director for Interface(Bob->Alice) (reversed).
        ^-- Charlie Director for Interface(Charlie->Bob) (reversed).
```

Supposing Dave cannot forward the packet and needs to send an error, he does not know where Charlie's Director ends and Bob's begins so he can't re-order them but because they are bit reversed, he can reverse the order by bit reversing the entire Route Label.

Route Label after bit reversal:

```
010110 110101 1000 0000000000000000000000000 0000001 10011 110011011
                                             ^^^^^^^ ^^^^^ ^^^^^^^^^-- Charlie's Director for Interface(Cha
                                                ^^^^ ^-- Bob's Director for Interface(Bob->Alice).
                                                   ^ Alice's Director for her Self Interface.
```

Dave can then send the packet back to Charlie who need not know what it is in order to forward it correctly on to Bob and then to Alice. If the packet had reached Fred, he would be able to use the same technique of reversing the Route Label in order to determine its origin.

Route Label Manipulations

Despite a Route Label's relative opacity, there are still certain functions which can be carried out on Labels.

Splice

The Splice operation takes a Route Label from pointA to pointB and concatenates it with a Label from pointB to pointC yielding a Route Label for a route from pointA to pointC.

Splicing is done by XORing the second part with `1` and shifting it left by the log base 2 of the first part, then XORing the result with the first part.

Given:

```
routeAB =        0000000000000000000000000000000000000000000001011101110101011001

routeBC =        0000000000000000000000000000000000000000000000000000110101010100
XOR 1            0000000000000000000000000000000000000000000000000000000000000001
equals           0000000000000000000000000000000000000000000000000000110101010101

shift left by the log base 2 of routeAB
                 0000000000000000000000000000000000110101010101000000000000000000
XOR  routeAB     0000000000000000000000000000000000000000000001011101110101011001
equals routeAC   0000000000000000000000000000000000110101010100011101110101011001
                                                              ^-- Note the overlap bit
```

The log base 2 represents the index of the first set bit, starting from the right. This means that shifting by the log base 2 leaves 1 bit of overlap, this along with the XORing of the second part (`routeBC`) against `1` causes the highest bit in the first part to be overwritten.

Unsplice

The inverse of the splice operation converts a full Route Label to a representation which would be useful to a node along the path. That is to say from routeAC and routeAB it derives routeBC.

```
routeAC =        0000000000000000000000000000000000110101010100011101110101011001
routeAB =        0000000000000000000000000000000000000000000001011101110101011001

routeBC = routeAC shifted left by the log base 2 of routeAB

routeBC          0000000000000000000000000000000000000000000000000000110101010100
```

RoutesThrough

Given two routes, it is possible to determine whether one route is an extension of another one, this is similar to the reverse of the the splicing routine. To determine whether routeAC "routes through" the node at the end of by routeAB, one simply removes the most significant 1 in routeAB, trims down the left side of routeAC so that both routes are the same length, and compares the results.

```
routeAC =        0000000000000000000000000000000000110101010100011101110101011001

routeAB =        0000000000000000000000000000000000000000000001011101110101011001

Trimmed to the same length:

routeAC'         0000000000000000000000000000000000000000000000011101110101011001
routeAB'         0000000000000000000000000000000000000000000000011101110101011001
```

Equality indicates routeAC routes through the node at the end of routeAB.

Encoding Schemes

Because the conditions under which the switches operate may differ dramatically, the actual Encoding Scheme is left as an implementation detail. Some switches may be devices with 15 physical ports wherein a 4 bit fixed width Encoding Scheme would be wise, other switches may be in wireless networks where the number of reachable cjdns nodes, and thus the number of Interfaces may grow and shrink. These devices may prefer a variable width encoding in order to save Label space without sacrificing expandability. There are however a few limitations placed upon encoding methods in order to allow the above manipulations.

Self Interface Director

In order for Route Labels to be able to be spliced together, the most significant bit in a Label must always be `1` so that we know where it ends. Since every route ends with a Self Interface, the Director which represents the Self Interface must always be encoded as `1` with 3 or more leading zeros. Furthermore, a node must never send a packet with a Route Label for which the highest 3 bits are not zero. This is important so that the reverse route data applied by routers along the path is not mistaken for additional forward route.

Finally, a switch should alias any Director ending with `0001` to the Self Interface Director. For example: a switch using a 6 bit fixed width Encoding Scheme would have to alias `010001` , `100001` and `110001` to the Self Interface Director `000001` . The reason for this is because an unsuspecting router might splice a valid path which uses every bit of the label space such that the most significant three bits of the entire Route Label are `0001` . In this case the bits added by the first switch will be right up against the Self Interface Director and could be mistaken for another forward Director.

Variable Width Encoding

If a switch implementation uses variable width encoding, it's obvious that some Interfaces will get short Directors while other Interfaces will be stuck with the longer Directors. Less obvious is what happens when a packet coming from a "small" Interface is sent to a "big" Interface. The switch shifts off the big Director leaving a big empty space on the left. To put only a small Director in that space would not work because there would be unaccounted for space in the reverse route which would cause a failure at the next switch in the path.

In order to make it work, small Interfaces need to be able to be represented using big Directors. For any size, all smaller sizes need to be able to be represented.

In the other direction, a packet coming from a big Interface with only a small next Director must be dropped (and an error sent back) because the return route simply cannot be represented. To prevent this problem, when a node on the far end of a big Interface inquires about a node behind a small Interface, the first Director in the Route Label must be converted by the router to the big form before the packet is sent.

A still more subtle problem with variable width encoding is that a Label for the route between nodeA to nodeB may be different depending on how one reached nodeA. Since this would be a severe hindrance to efficient routing algorithms using path inference, a node is required to *describe* (through inter-router communication) how it encodes numbers.

Encoding Scheme Definition

In order to give routers the ability to chain paths from a graph of linked nodes while still preserving Variable Width Encoding, a node is required to describe its Encoding Scheme to other nodes and adhere to a few rules to make possible the conversion of a Director to a wider bit width by other nodes.

The Encoding Scheme Definition consists of a array of representations of the Encoding Forms allowed in that Encoding Scheme, each Encoding Form representation having the following three fields:

- **prefixLen**: The number of bits in the Director Prefix. This is represented on the wire as a 5 bit integer.
- **bitCount**: The number of bits in the Director, not including the Director Prefix. This is represented on the wire as a 5 bit integer.
- **prefix**: The Director Prefix used in this Encoding Form, this is represented on the wire using as many bits as are given in **prefixLen**. This value must be the same as the prefix which will be seen in the Director in the label but since it is opaque, no specific byte order is defined.

The Encoding Scheme Definition is represented on the wire as a concatenation of Encoding Forms. Each form is represented in *reverse* the order given above with **prefix** furthest to the left, followed by **bitCount** then **prefixLen** furthest to the right. It is sent over the wire in *little endian* byte order allowing the buffer to be read and written forward while the data is unpacked from least significant bit to most significant bit.

Conversion between Encoding Forms

Just telling other nodes how numbers are encoded is not enough to allow inference of paths. As routers chain together the Route Labels for paths between different nodes, they need to *alter* the Encoding Form for Directors in the Route Labels to avoid crafting a Route Label for which the reverse path is not expressible.

The two types of alteration which need to be allowed are *extension* and *truncation*. Given a route between nodeX and nodeY, a node must be able to *extend* the first Director in that route by changing it from one of the forms advertised by nodeX to another. This involves changing the Director Prefix and then extending the non-prefix section of the first Director by adding zero bits to the left side. *Truncation* is much the opposite, one would convert the Director from a wider form to a narrower by changing the Director Prefix and truncating zero bits from the left side.

In order to remain compatible with these optimizations, a switch must disregard a change in the number of most significant zero bits in its Director, in the case of multi-byte Directors this may influence byte order decisions. Obviously a switch implementor must design their Encoding Scheme so that a Director is unambiguous and cannot be confused with the least significant bits of the data but since this is a requirement for a successful switch implementation, it goes without saying.

In Memory Representation

While the switch protocol is inherently linked to the underlying carrier, there are certain expectations made by the protocols above the switch layer about how a switch header will appear in memory.

```
                   1               2               3
     0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  0 |                                                               |
    -                          Route Label                          -
  4 |                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  8 |      Type     |                   Priority                    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The `Route Label` field unsurprisingly holds the 8 byte `Route Label`, the `Type` field indicates the type of packet. Reserved packet types are `0` for opaque data, and `1` for switch control messages (eg errors). The `Priority` field is reserved for Quality of Service purposes, in the current implementation it is always zero.

The Router

A router has 3 functions, it periodically searches for things, responds to searches, and forwards packets. When a router responds to a search, it responds with nodes which it thinks will get closer to the destination. The responses MUST NOT have addresses which are, in address space distance, further from the responding node than the search target, and they MUST NOT have routes which begin with the same interface as the route to the querying node. These two simple rules provide that no search will ever go in circles and no route will ever go down an interface, only to be bounced back. While the second rule can only be enforced by the honor system, querying nodes MUST double check the first rule. The node doing the searching adds the newly discovered nodes to their routing table and to the search, then continues the search by asking them.

Upon receiving a search response containing one's own address, a node SHOULD purge all entries from its table whose routes begin with that route. This will control the proliferation of redundant routes.

The "address space distance" between any two given addresses is defined as the of the result of the two addresses XOR'd against one another, rotated 64 bits, then interpreted as a big endian integer. The so called "XOR metric" was pioneered in the work on Kademlia DHT system and is used to forward a packet to someone who probably knows the whole route to the destination. The 64 bit rotation of the result is used to improve performance where the first bits of the address is fixed to avoid collisions in the IPv6 space.

Adding nodes to the routing table from search responses is done by splicing the route to the node which was asked with the route to the node in the response, yielding a route to the final destination through them.

Router messages are sent as normal UDP/IPv6 packets except that their UDP source and destination port numbers are zero and the hop limit (TTL) field in the IPv6 header is set to zero. Any packet which meets these characteristics is to be considered a router message and any packet which doesn't is not. It is critical that inter-router communications are themselves, not routed because it would break the label splicing for search responses.

The content of the inter-router messages is bEncoded dictionaries. Routers send search queries which have a key called "q", and replies which don't. Routers SHOULD reply to a message with a "q" entry but MUST NOT reply if there is none, lest they reply to a reply. All messages have a transaction id number, a sort of cookie made of a bencoded string containing arbitrary bytes which must be reflected back in the reply. The most common query is a find node or "fn" query. "fn" queries have a field called "tar" for the target address which the node is looking for. Responses to "fn" queries have a field called "n" which is a binary string containing the 32 byte public keys and 8 byte switch labels for the responses.

Example fn query in JSON:

```
{
    "q":    "fn",
    "tar":  "abcdefghhijklmno",
    "txid": "12345"
}
```

Same query bEncoded as the routers use:

```
d1:q2:fn3:tar16:abcdefghhijklmno4:txid5:12345e
```

Example fn reply in JSON: NOTE: this reply only shows 2 nodes returned and is for illustration purposes in most cases the number would be an implementation specific constant around 8.

```
{
    "n": "cdefghijklmnopqrstuvwxyzabcdefghi1234567qponmlkjihgzyxwvutsrstuvwxyzabcde2345678"
    "txid": "12345"
}
```

Same reply bEncoded

```
d1:n80:cdefghijklmnopqrstuvwxyzabcdefghi1234567qponmlkjihgzyxwvutsrstuvwxyzabcde2345678e
```

The nodes in an fn reply are ordered from worst to best so the best answer is the last entry in the reply.

Routers choose the node to forward a packet to in a similar way to how they answer search queries. They select nodes from their routing table except in this case the selection contains only one node. The packet is sent through the CryptoAuth session corresponding to this node and the label for getting to it is applied to the packet before sending to the switch. The "search target" for forwarding a packet is the IPv6 destination address of the packet.

The CryptoAuth

The CryptoAuth is a mechanism for wrapping interfaces, you supply it with an interface and optionally a key, and it gives you a new interface which allows you to send packets to someone who has that key. Like the rest of cjdns, it is designed to function with best effort data transit. The CryptoAuth handshake is based on piggybacking headers on top of regular data packets and while the traffic in handshake packets is encrypted and authenticated, it is not secure against replay attacks and has no forward secrecy if the private key is compromised. The CryptoAuth header adds takes 120 bytes of overhead to the packet, causing a fluctuating MTU.

There are 5 types of CryptoAuth header:

1. Connect To Me - Used to start a session without knowing the other node's key.
2. Hello Packet - The first message in beginning a session.
3. Key Packet - The second message in a session.
4. Data Packet - A normal traffic packet.
5. Authenticated - A traffic packet with Poly1305 authentication.

All CryptoAuth headers are 120 bytes long except for the Data Packet header which is 4 bytes and the Authenticated header which is 20 bytes. The first 4 bytes of any CryptoAuth header is a big endian number which is used to determine its type, this is the so-called "Session State" number. If it is the inverse of zero, it is a Connect To Me header, if it is zero, it is a Hello Packet, if one or two, it is a Hello Packet or repeated Hello Packet, if it is three or four, it is a Key Packet or repeated Key Packet. If it is any number larger than four, it is either a Data Packet or an Authenticated packet, depending on whether authentication was requested during the handshake.

Handshake packet structure:

```
                     1               2               3
       0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    0 |                         Session State                         |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    4 |                                                               |
      +                                                               +
    8 |                         Auth Challenge                        |
      +                                                               +
   12 |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   16 |                                                               |
      +                                                               +
   20 |                                                               |
      +                                                               +
   24 |                                                               |
      +                         Random Nonce                          +
   28 |                                                               |
      +                                                               +
   32 |                                                               |
      +                                                               +
   36 |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   40 |                                                               |
      +                                                               +
   44 |                                                               |
      +                                                               +
   48 |                                                               |
      +                                                               +
   52 |                                                               |
      +                     Permanent Public Key                      +
   56 |                                                               |
      +                                                               +
   60 |                                                               |
      +                                                               +
   64 |                                                               |
      +                                                               +
   68 |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   72 |                                                               |
      +                                                               +
   76 |                                                               |
      +                     Poly1305 Authenticator                    +
   80 |                                                               |
      +                                                               +
   84 |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   88 |                                                               |
      +                                                               +
   92 |                                                               |
      +                                                               +
   96 |                                                               |
      +                                                               +
  100 |                                                               |
      +          Encrypted/Authenticated Temporary Public Key         +
  104 |                                                               |
      +                                                               +
  108 |                                                               |
      +                                                               +
  112 |                                                               |
      +                                                               +
  116 |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                                                               |
      +        Variable Length Encrypted/Authenticated Content        +
      |                                                               |
```

1) Connect To Me Packet

If "Session State" is equal to the bitwise complement of zero, the sender is requesting that the recipient begin a connection with him, this is done in cases when the initiator of the connection does not know the key for the recipient. If the entire header is not present the recipient MUST drop the packet silently, the only field which is read in the packet is the "Permanent Public Key" field, all others SHOULD be ignored, specifically, content MUST not be passed on because it cannot be authenticated. The recipient of such a packet SHOULD send back a "hello" packet if there is no established connection. If there is already a connection over the interface, the recipient SHOULD NOT respond but MAY allow the connection to time out faster.

2) Hello Packet

If the "Session State" field is equal to the one or two, the packet is a Hello Packet or a repeated Hello Packet. If no connection is present, one MAY be established and the recipient MAY send a Key Packet in response but it is RECOMMENDED that he wait until he has data to send first. A node who has sent a Hello Packet, has gotten no response and now wishes to send more data MUST send that data as more (repeat) Hello Packets. The temporary public key and the content are encrypted and authenticated using the permanent public keys of the two nodes and "Random Nonce" in the header. The content and temporary key is encrypted and authenticated using crypto_box_curve25519poly1305xsalsa20() function.

3) Key Packet

If the "Session State" field is equal to two or three, the packet is a Key Packet. Key Packets are responses to Hello Packets and like Hello Packets, they contain a temporary public key encrypted and authenticated along with the data. Once a node receives a Key Packet it may begin sending data packets. A node who has received a Hello Packet, sent a Key Packet, gotten no further response, and now wishes to send more data MUST send that data as more (repeat) key packets.

4) Data Packet

The traditional data packet has only 4 bytes of header, these 4 bytes are the nonce which is used for the cipher, the packet is enciphered using crypto_stream_salsa20_xor() with the nonce, converted to little endian encoding, and copied to the first four bytes of the 8 byte nonce required by crypto_stream_salsa20_xor() unless the node is the initiator of the connection (the sender of the hello packet), in which case it is copied over the second four bytes of the space, thus allowing for a single session to handle 2^32 packets in either direction.

5) Authenticated Packet

The Authenticated Packet is sent if Poly1305 authentication was requested by either node during the handshake. Like the Data Packet, the first 4 bytes is used as the nonce, in this case it is a 24 byte nonce and crypto_box_curve25519poly1305xsalsa20() is used to encrypt and decrypt the data, but the methodology is exactly the same. If a packet is not authenticated, it MUST be silently dropped.

ReplayProtector

When packet authentication is enabled, the packet is checked for replay attacks (intentional or accidental) the replay protection method is to use a 32 bit offset and a 32 bit bitfield to create a sliding window. When a packet comes in, its nonce is compared to the offset, if it is less then the offset, it is discarded. If when subtracted from the offset, the result is less than or equal to 32, 1 is shifted left by the result, bitwise ANDed against the bitfield and compared to zero, if it is not zero then the packet is a duplicate and is discarded. If it is zero then it is OR'd against the bitfield to set the same bit is set and the packet is passed along. If the result of subtraction is greater than 32, 32 is subtracted from it, this result is added to the offset, the bitfield is shifted left by this amount, then the least significant bit in the bitfield is set. This is obviously only available when packets are authenticated but provides a secure protection against replay attacks and accidentally duplicated packets EG: from 802.11 noise.

This solution is limited in that packets which are more then 32 "slots" out of order will be discarded. In some cases, this could be a benefit since in best effort networking, never is often better than late.

Authentication field:

This field allows a node to connect using a password or other shared secret, the AuthType field specifies how the secret should be used to connect.

```
                   1               2               3
     0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  0 |   AuthType    |                                               |
    +-+-+-+-+-+-+-+-+        AuthType Specific                      +
  4 |                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  8 |A|                      AuthType Specific                      |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The "A" flag is used to indicate that the node is requesting the session use Poly1305 authentication for all of its packets. The "AuthType Specific" fields specific to the authentication type.

AuthType Zero

AuthType Zero is no authentication at all. If the AuthType is set to zero, all AuthType Specific fields are disregarded and SHOULD be set to random numbers.

AuthType One

AuthType One is a SHA-256 based authentication method.

```
                   1               2               3
     0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  0 |   Auth Type   |                                               |
    +-+-+-+-+-+-+-+-+           Hash Code                           +
  4 |                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  8 |A|        Derivations          |           Additional          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

With AuthType One, the shared secret (password) is hashed once and the result is appended to the 32 byte output from scalar multiplication of the curve25519 keys these 64 bytes are hashed again with SHA-256 to make the symmetric key to be used for the session. It is also hashed a second time and the result copied over the first 8 bytes of the authentication header before the AuthType field is set. The effect being that the "Hash Code" field contains bytes 2 through 8 the hash of the hash of the password. This is used as a sort of username so that the other end knows which password to try using in the handshake.

If Derivations is non-zero, an additional step is included, the two most significant bytes of the password hash are XORed against the two bytes of the network representation of Derivations and it is hashed using SHA-256 once again before being included in the generation of the symmetric key. This form is notably NOT used in the Hash Code field.

This allows a node Alice, to give a secret to Charlie, which he can use to start a CryptoAuth session with Bob, without leaking Alice's shared secret. This allows nodes to generate, share and derive secrets through trusted connections, creating new trusted connections and use them to share more secrets, adding a measure of forward secrecy in the event of a cryptographic weakness found in the asymmetric cryptography.

Pulling It All Together

The journey of a packet begins at the user interface device (TUN or similar). The user sends an IPv6 packet which comes in to the TUN device and enters the engine, it is checked to make sure its source and destination addresses are valid and then a router lookup is made on the destination address. cjdns addresses are the first 16 bytes of the SHA-512 of the SHA-512 of the public key. All addresses must begin with the byte `0xFC` otherwise they are invalid, generating a key is done by brute force key generation until the result of the double SHA-512 begins with `0xFC` .

After the router lookup, the node compares the destination address to the address of the next router, if they are the same, the inner layer of encryption is skipped. Assuming they are different, the IPv6 header is copied to a safe place and a CryptoAuth session is selected for the destination address, or created if there is none, and the packet content is passed through it. The IPv6 header is re-applied on top of the CryptoAuth header for the content, the packet length field in the IPv6 header is notably *not* altered to reflect the headers which are now under it.

The packet is now ready to send to the selected router. For sending the packet to the router, a CryptoAuth session is selected for the router's address and the packet, from IPv6 header down, is passed through it. A switch header is applied to the resulting encrypted structure and it is sent down to the switch for routing.

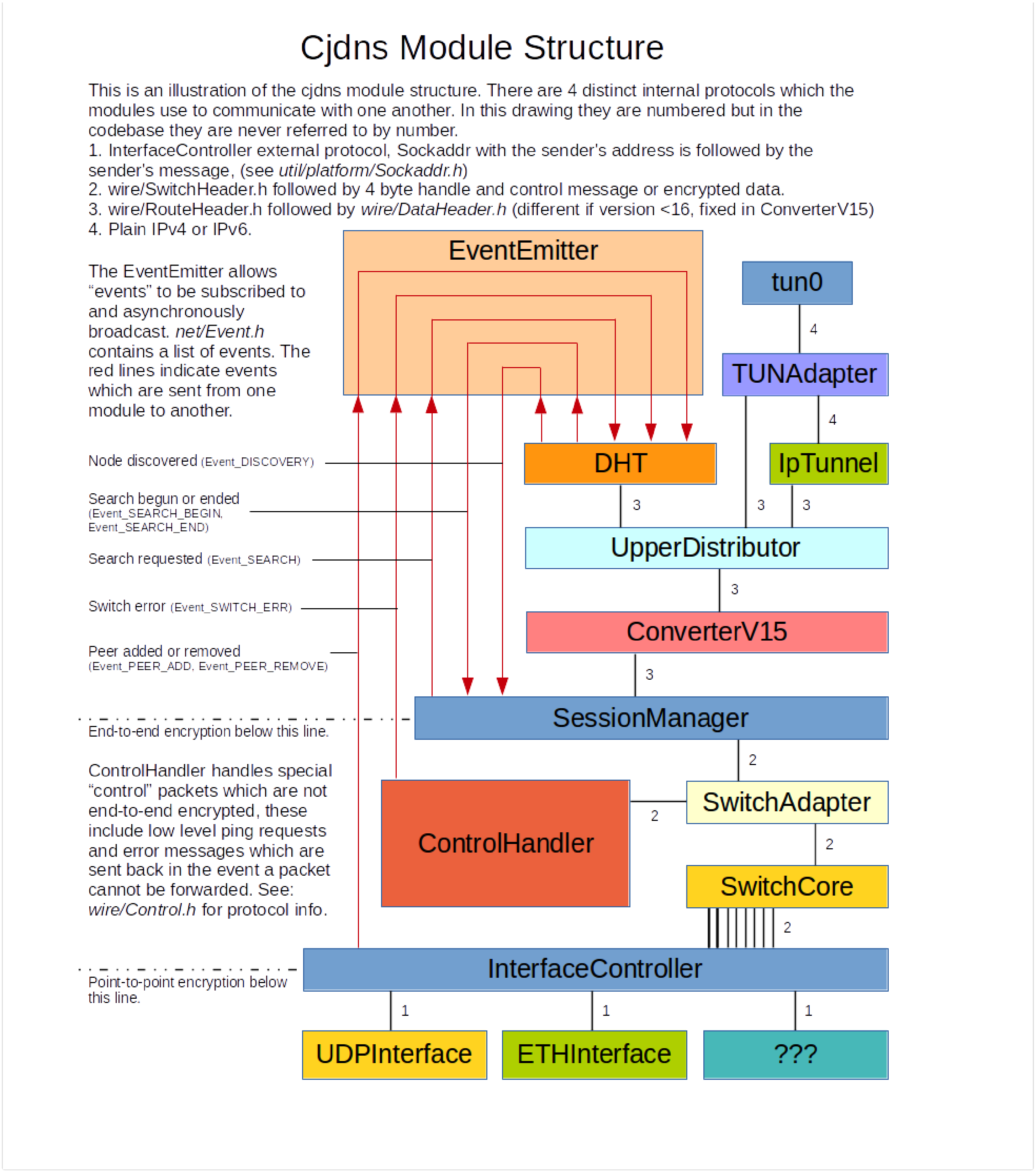
The switch takes the packet and sends it to a network module which uses yet another CryptoAuth session to encipher and authenticate the packet from the switch header down. The resulting data is packaged in a network packet and sent to the switch at the next node.

Upon receiving the packet, the next node sends the packet through its CryptoAuth session thus revealing the switch header and it sends the packet to its switch. The switch most likely will send the packet out to another endpoint as per the dictate of the packet label but may send it to its router, eventually the node for which the packet is destine will receive it.

The router, upon receiving the packet will examine it to see if it appears to be a CryptoAuth Connect To Me packet, Hello packet, or Key packet. If it is one of these, it will insert the IPv6 address, as derived from the public key in the header, into a hashtable so it can be looked up by the switch label. Otherwise it will do a lookup. If the Address cannot be found in its hashtable, it will try asking the router if it knows of a node by that label and if all fails, the packet will be dropped.

From the IPv6 address, it will lookup the CryptoAuth session or create one if necessary, then pass the opaque data through the CryptoAuth session to get the decrypted IPv6 header.

If the source address for the packet is the same as the double SHA-512 of the public key for the router from which it came, it's assumed to have no inner layer of encryption and it is written to the TUN device as it is. If its source address is different, it is passed back through a CryptoAuth session as selected based on the source IPv6 address. The IPv6 header is then moved up to meet the content (into the place where the CryptoAuth header had been) and the final packet is written out to the TUN device.



Cjdns Module Structure

This is an illustration of the cjdns module structure. There are 4 distinct internal protocols which the modules use to communicate with one another. In this drawing they are numbered but in the codebase they are never referred to by number.

1. InterfaceController external protocol, Sockaddr with the sender's address is followed by the sender's message, (see *util/platform/Sockaddr.h*)
2. wire/SwitchHeader.h followed by 4 byte handle and control message or encrypted data.
3. wire/RouteHeader.h followed by *wire/DataHeader.h* (different if version <16, fixed in ConverterV15)
4. Plain IPv4 or IPv6.

The EventEmitter allows "events" to be subscribed to and asynchronously broadcast. *net/Event.h* contains a list of events. The red lines indicate events which are sent from one module to another.

ControlHandler handles special "control" packets which are not end-to-end encrypted, these include low level ping requests and error messages which are sent back in the event a packet cannot be forwarded. See: *wire/Control.h* for protocol info.

EventEmitter
tun0
4
TUNAdapter
4
Node discovered (Event_DISCOVERY)
DHT
IpTunnel
Search begun or ended (Event_SEARCH_BEGIN, Event_SEARCH_END)
3
3
3
UpperDistributor
Search requested (Event_SEARCH)
3
Switch error (Event_SWITCH_ERR)
ConverterV15
Peer added or removed (Event_PEER_ADD, Event_PEER_REMOVE)
3
SessionManager
End-to-end encryption below this line.
2
ControlHandler
2
SwitchAdapter
2
SwitchCore
2
InterfaceController
Point-to-point encryption below this line.
1
1
1
UDPInterface
ETHInterface
???