Squawk (AWKzine Issue #1)

Todd Coram (todd@maplefish.com)

February 13, 2026

A Simple Secure Messaging Client and Server


```
BEGIN { VERSION="Squawk Version 1.51" }
```

This file is a (semi) Literate Programming Document of Squawk. [2] Squawk is a cryptographically secure messaging client and server (all in one file) written in gawk (using AES256 encryption provided by `openssl`). Squawk is documented using Knit and AFT [3].

Contents

[1] To view a copy of the license, visit [http://creativecommons.org/licenses/by-sa/4.0

[2] The single file source is likely to be found at http://www.maplefish.com/todd/squawk.awk . This document can be found at http://www.maplefish.com/todd/squawk.pdf and http://www.maplefish.com/todd/squawk.html

[3] Knit is available at http://www.maplefish.com/todd/knit.html) and AFT is at http://www.maplefish.com/todd/aft.html. If you run Ubuntu then you may find yourself able to install AFT using apt install aft

1 Front Matter: What do I do with this?

This file is meant to be read. You are doing that now! You may be reading the source code (squawk.awk) or reading this as an HTML page or even as a PDF.

To generate the HTML page (squawk.html) of this file run this comand:

```
$ gawk -f knit.awk squawk.awk >squawk.aft && aft squawk.aft && aft squawk.aft
```

(Yes, yes... run `aft` twice, it needs a 2nd pass to build up a table of contents...)

The squawk.awk file can also be run just like any normal gawk script.

For example, to run as a server:

```
$ gawk -M -f squawk.awk todd,bob,sue 6666
Running Server on Port  6666
White List = todd,bob,sue
Your Authentication Token is: iPO1ICJ4
Serving..
```

Now the server is waiting for client connections.

Here is a client example of sending a message (from bob to todd):

```
$ gawk -M -f squawk.awk localhost:6666 iPO1ICJ4 send todd bob "Hello there!"
OK
$
```

Bob's message is saved on the server, waiting for him to request it. For example, he may be listening for messages:

```
$ gawk -M -f squawk.awk localhost:6666 iPO1ICJ4 listen bob
```

As messages arrive, they will be printed out.

There are a couple more commands and many more options. You can see them by requesting help:

```
$ gawk -M -f squawk.awk help
```

You can jump ahead to Client and Server Command Line Parsing (§ 3.2) for the gory details.

This may look cumbersome, but the idea is to have a simple command line interface, not a GUI or website. You can certainly build a website or GUI to wrap this command. That is left as an excercise for the reader.

2 Concept (or Why?)

The concept is simple: What is the simplest, yet secure, easy to deploy, text messaging service that could work? What if you want zero configuration, zero dependencies and zero footprint? What if you want to just stand up an *adhoc* single use chat server?

Welcome to Squawk.

Squawk is a reasonably secure text messaging service that anyone (who has `gawk` and `openssl` and a POSIX compliant `shell` available on their computer/laptop/tablet/phone) can stand up and run.

Here lies is a single script (under 500 lines of code) that can function as either a client or server, with no hidden *magic*. You just need access to a command line shell and a (cloud) server that has an open UDP port.

Outside of key generation and encryption provided by `openssl`, everything is in this single file that can be executed by the `gawk` interpreter (which is probably already installed on your Linux/BSD/MacOSX system or can be easily installed). This includes the networking, command processing and authentification code. It's all implemented in gawk with no external libraries (or configuration files).

This makes this messaging system highly portable. Get a trusted copy of this file, and you can quickly start securely messaging.

That's it. While I make no claims to be truly secure Caveat emptor (§ 3.12)! but this is damn simple and likely safe for playing around with.

The server makes no use of disk storage and outputs no user information. It's all kept in RAM and goes away when you terminate.

All external (network) communication is with users (this script running as a client) using a single shared cryptographic key. [4]

This server does support the concept of *white lists* (limiting sending and receiving to a select set of users). White lists are mainly used to help prevent a user from cluttering memory by sending to mailboxes that are not ever going to be read.

2.1 Dependencies

Did I mention that this single file script has few dependencies? Well, here they are:

openssl - Just the command line binary, no libs. Used for key generation.

gawk - Unfortunately, this file isn't runnable by the true awk or mawk or any other variants. It utilizes the networking and *bignum* capabilities of GNU awk. If your system doesn't have gawk, you can more than likely easily find a version in your distro's official repo.

Read this file and you've reviewed everything used.

2.2 Adhoc Messaging

This is not a system you stand up long term on a server. It doesn't scale and has few provisions for surviving denial of service attacks.

The server is meant to be run *adhoc* (and not as a permanent service). It maintains no state and keeps everything in RAM. Stand it up; use it; shut it down.

But where can you run this server? Pick a cloud provider. I like Vultr, but there are others out there. Vultr is inexpensive and lets me run OpenBSD instances. I prefer to keep things simple. Whatever you pick, you won't need a lot of services or anything other than the above dependencies.

2.3 Gawk? Why Gawk?

I'm a long time Awk programmer. I've been using Awk since I first discovered **The Awk Programming Language** book back in 1988. It was (and still is) perhaps one of the best books on programming I've read. [5] It is also a powerful, interesting and underpreciated scripting language.

Unfortunately, plain old Awk doesn't have networking, bidirectional pipes and bignums, all of which as used copiously here. So, I turned to Gawk (GNU Awk).

Gawk isn't installed everywhere [6] but it is usually easily installed on most OSes.

The following Gawk extenstions are used here:

[4] This current version does not support "private" keys (between individual users) but it is planned for the future. The idea is to use this as either an adhoc 2 user one on one chat or as an adhoc *group* chat where everyone is sharing the same key. Group chats are currently singular and the group is called *all.*.

[5] Perhaps to be supplanted by the newer edition that came out in 2023.

[6] Not the default Awk in OpenBSD, which is the OS I am developing Squawk under.

1. Bignums (arbitrary precision numbers) are used for `SipHash` authentication which want 64 bit integers. Gawk (and Awk) usually use IEEF floats and at best will support up to 53 bit values.
2. UDP Networking
3. Bidirectional piping (e.g. reading and writing to subprocesses). Awk does only single direction.

Openssl is also fairly ubiquitous. We use the `openssl` command here, not the libraries.

I considered Perl at one point, but I like the simplicity of Awk. Awk is a POSIX standard and an essential part of things related to *UNIX* (e.g. MacOSX, *BSD and Linux). Gawk follows mostly. I'm not using Gawk features that are new, but ones that have been around for years and likely won't change. [7]

2.4 About the choice of UDP...

Yes, I know. UDP is, by definition: Unreliable. You want your messaging to be reliable. You don't want to miss a message or have it dropped because UDP packets are being discarded. TCP is what makes the most sense, but here are a couple of things about TCP that can make things a little more complicated:

1. TCP is connection based. You have to write a server that can handle multiple connections. That means network programming (e.g. states, sessions, etc) gets more complicated. Complicated beyond the capabilities of Gawk.
2. TCP is more resource intensive. Related to above: how do you handle clients that that "hog" connections and won't let go.

Squawk is not trying to be an enterprise class messaging system. It is meant primarily to be instructional and *interesting*.

That said, read on and you'll see how we can do reliable messaging using UDP without re-implementing TCP. We follow a very simple request/response pattern that will make it evident when packets **are** dropped. [8]

2.5 Have Fun and Stay Safe

Can you trust Squawk with transfering sensitive (private) information? No. Can you trust Squawk to prevent spoofers and interlopers? No. This was developed for fun and to prove that you can indeed implement a simple "Hey, can you buy milk on your way home?" style of Internet messaging system without buying a service or trusting someone else's system or trying to make heads or tails out of some complicated open source build.

This is about demystifying things that shouldn't be *magical*. In fact, given all the limitations and (potential) vulnerabilities of Squawk may give you new found respect for those folk who **do** build scalable and secure messaging systems.

So, with that out of the way, let's dive in!

3 Implementation

3.1 Configurable Things

Here lies things you may want to configure and change (just for the server).

But, before we begin, let's do a hack to try and figure out if the -M flag was supplied (to force gawk to use arbitrary precision (bignum) – which is needed for our implementation of SipHash. (This was a *found* hack, not mine.0

```
BEGIN {
    Err = ""
    if (2^-2^2^2 == 0) {
        Err = "Please use -M; i.e.  $ gawk -M ..."
        exit 1
```

[7] This is one of the reasons I didn't choose to implement this in Python. Which Python is installed? Which batteries are included?

[8] UDP packet delivery order isn't a concern if we keep messages smaller than the maximum size of a UDP packet, hence our suggestion in Messaging Protoco (§ **??**) to keep message small.

```
    }
}
```

`Port` is the port the server listens to requests on. You'll want to pick something that your cloud provider will let you connect to. We use UDP, so you'll need one configured to allow UDP requests to be received. Luckily, this shouldn't be too big of a deal.

```
BEGIN {
    Debug = 0 # set to 1 to enable debugging output
    Port = 8080  # UDP, not TCP
    Max_Mbox_Messages = 50
}
```

That's it. You don't need to configure anything else.

3.2 Client and Server Command Line Parsing

Using this script as a server is different than as a client. Let' s deal with those differences by parsing the command line parameters.

Generally, the server generates a `Auth_Token` and knows its own `Host` and as above, can define its own `Port` and collect a comma separated list of clients that the application will be restricted to into `White_List_String`.

```
BEGIN {
    if (ARGC > 1 && ARGV[2] == /help/) {
        Err = VERSION \
         "\nServer Usage:\n" \
        "  $ gawk -M -f squawk.awk [-- -t  token] [whitelist] [port]\n" \
        "\nClient Usage:\n" \
        "  $ gawk -M -f squawk.awk <Host>[:port]  <token> send <to>|all <from> '<message>'\n" \
        "  $ gawk -M -f squawk.awk <Host>[:port]  <token> check <mbox> '<message>'\n" \
        "  $ gawk -M -f squawk.awk <Host>[:port]  <token> listen <mbox> '<message>'\n"
        exit 1
    }

    aidx = 1
    if (ARGV[aidx]  == "-t") {
        Auth_Token = ARGV[aidx+1]
        aidx += 2; ARGC -= 2
        Server = 1
    }

    Server = (ARGC < 4) ? 1 : 0
    if (Server)  parse_server_options(aidx); else  parse_client_options()
    fflush()
}
```

3.2.1 Server Command Line Parsing

As a convenience, you can supply you own variable length token by supplying `- -t token` as the first argument. (Note the two dashes, as this is required by gawk so that `-t` isn't interpreted as a gawk parameter.)

Be careful here. The main motivation here is if you want to *pre-share* a token before running the server or you had to stop the server briefly before restarting it and have already given out the token.

```
function parse_server_options(aidx) {
    print ("Server", ARGC, aidx, ARGV[aidx], ARGV[aidx+1])
    if (ARGC ==3) {
        # We have a whitelist and port number
```

```
        White_List_String = ARGV[aidx]
        Port = ARGV[aidx+1];
    }
    else if (ARGC == 2) {
        # We have either a whitelist or a port number
        #
        if (int(ARGV[aidx]) > 0) Port = ARGV[aidx]; else  White_List_String = ARGV[aidx]
    }
    print("Running Server on Port ", Port)
    if (White_List_String) {
        print("White List=", White_List_String)
        split(White_List_String, Wa, ",")
        for (i in Wa) { White_List[toupper(Wa[i])] = 1 }
        delete Wa
    }
    Host_Listener = "/inet/udp/" Port "/0/0"
}
```

3.2.2 Client Command Line Parsing

The client must supply more arguments. In particular, the `Host`, the `Auth_Token`, a command and optionally a `Port` (supplied as `Host:Port`).

```
function parse_client_options() {
    Host = ARGV[1]
    if (Host ~/:/) {
        split(Host, hp, ":")
        Host = hp[1]
        Port = hp[2] + 0
    }
    Host_Addr = "/inet/udp/0/" Host "/" Port
    Auth_Token = ARGV[2]
    Cmd = ARGV[3]
    for (i=0; i < ARGC-4; i++)  Cmd_ARGV[i] = ARGV[4+i]  # Save args for later..
    ARGC = 0
}
```

But, what about access control? How do you make sure that unauthorized people don't start using and abusing your message server?

3.3 Authorization

Authorization is done *out of channel*. When you run the server it generates a random key. You will want to write down this key and pass it to your friends. This is what they will need to communicate with the server. This seeds both the encryption and authentication. Any packets received. by the server, that doesn't decrypt or authenticate correctly are rejected.

There is a trade off between having a nice long secure password (that is hard to remember and type) and a short one (that is easy to pass to transcribe – say, like over a voice call or SMS) that isn't so secure. We err on the side of short and easy.

A digression (but an important one). We are going to secure this server with AES256 in CTR mode with a 16 byte SipHash (§ 3.10.2) for authentication. This is... overkill, but is pretty damn secure. That said, trying to remember and pass around 256 bit (16 byte) keys is a pain in the ass. So, we are going to deal with short "tokens" that are, internally, hashed into sha256 digests.

What all this means is that at the end of the day we are only as secure as the token itself. A longer token is better than a shorter one.

Rather than get too complex, we are going to keep this first stab at token generation very basic and boring: a 6 byte number (2^48) represented as an 8 digit base64 string. This yields a fairly large space for randomization and thus should be hard to guess and hard for an attacker to brute force (in the presumably short amount of time you should be keeping the server running).

```
BEGIN {
    if (Server) {
        if (Auth_Token == "") {  # if a token wasn't supplied, generate one.
            "openssl rand -base64 6" | getline Auth_Token
        }
        print("Your Authentication Token is:", Auth_Token)
        fflush();
    }
    Auth_Key = gen_key(Auth_Token)
}
```

All messages sent to and replied from the server is encrypted with the `Auth_Key` (and a counter or "nonce" that is never repeated). You share the authorization token that the server prints out when started and a key is generated from this.

```
function gen_key(token) {
    "echo " token "|openssl dgst -sha256" | getline
    return $2
}
```

3.4 Server Protocol

The protocol for the server is inefficient and simple We don't care about inefficiency here. We care about simplicity.

Awk isn't great at handling binary data, so we will be encoding everything as text. This means that when it reads an incoming UDP packet it will be all text values in a nice format that awk can deal with. So...

```
[request or response]  [MAC] [CTR]
```

where `request or response` is the encrypted message (encoded in base64), `MAC` is the authentication code (encoded in hex) and CTR (encoded in hex) is a unique number that is never repeated. This unique number is cryptographically combined with the shared key.

All fine, but how does the CTR get generated? For our use, the CTR is seeded with a 96 bit randomly generated IV.

For AES256 CTR Mode: `CTR = IV + N` and encryption is: `C = P xor AES_Encrypt(CTR`, where P is plaintext and N is our counter.

This is one of the reasons we use the 'bignum" (-M) capability of gawk. Our CTR counts up forever! (well, a very large number at least). We randomly generate our IV (the basis for the CTR) using `openssl`.

Since every client (and server) is generating their on IV/CTR, there is a possibility of overlapping (which is very bad from a crypto perspective). However, given the 0 -> 2^128 space of starting numbers, and a few dozen (or hundreds) of messages the probability is fairly low. We follow the standard practice of allocating a 96 bit IV with 32 bits left over for the CTR.

```
BEGIN {
    "openssl rand -hex 12" | getline IV_S
    IV = strtonum("0x" IV_S "00000000")
}
```

Both the server and client must increment the CTR by the number of blocks encrypted since `openssl` does this internally (without letting us know).

```
function incr_IV_counter(cnt) {
    IV += cnt
}
```

3.5 Starting up Client or Server

Here is the last thing we do... we run either as a server or client.

```
END {
    if (Err != "") {
        print Err >"/dev/stderr"
        exit 1
    }
    if (Server) {
        print "Serving..."
        message_server()
    } else {
        client_request(Cmd, Cmd_ARGV)
    }
}
```

3.6 The Server Details

Let's jump right into the server. This server can only handle one client at a time. It should handle them pretty quick, but all transactions are single queued queries and responses. In gawk we do this via a single getline, the handling which may send something back to the client and a `close`, freeing up the process to handle another client.

```
function message_server(    msg, iv) {
    while (1) {
        Host_Listener |& getline
        iv= strtonum("0x" $3)
        handle_request($1, $2, iv)
        close(Host_Listener)
    }
}
```

This is where all incoming messages are decoded. We first filter out the bad messages and then determine what to do. If we are sending stuff to the client, we do so and be sure to increment our IV.

```
function handle_request(req, mac, iv,   pt, hash) {
    S_Tx_Cnt++
    pt = decrypt(req, Auth_Key, iv)
    hash = siphash(pt, Auth_Key_Bin, 16)
    if (hash !~ mac) {
        print "Bad auth" >"/dev/stderr"
        send_response("NAK")
        S_Bad_Auth_Cnt++
    } else {
        # print "Got a message" >"/dev/stderr"
        send_response(message_processor(pt))
    }
}
```

Here is how we send the reponse back to the client. Be sure to hex encode the IV.

```
function send_response(msg,  ec,mac) {
    ec =encrypt(msg, Auth_Key, IV)
    mac = siphash(msg, Auth_Key_Bin, 16)
    print(ec, mac, sprintf("%032x", IV)) |& Host_Listener;
    incr_IV_counter(int(length(ec)/16+1))
```

```
}
```

It may be useful to collect statistics on the server. How many users? Messages? is it being abused? We want to keep this server *ephemeral* so we don't want to keep too much info about users or how they are using the system.

Here are some stat variables:

```
BEGIN {
    S_Msg_In_Cnt = 0  # Mbox Messages incoming
    S_Msg_Out_Cnt = 0 # Mbox Messages outgoing
    S_Mbx_Full_Cnt = 0 # MBox full rsponses
    S_Bad_Req_Cnt = 0 # Number of unrecognized request commands
    S_Msg_Delivery_Acked_Cnt = 0 # Number of deliveries acked
    S_Tx_Cnt = 0 # Total transactions
    S_Bad_Auth_Cnt = 0 # Total "bad" transactions (failed auth)
    S_Mbx_Cnt = 0 # Total mailboxes created
    S_Bad_User_Cnt = 0 # Attempts to send to a non-white-list user
}
```

We can look at statistics anytime by calling this.

```
function dump_stats() {
    print""
    print("Mboxes\t:", S_Mbx_Cnt)
    print("Recvd Msgs\t:", S_Msg_In_Cnt)
    print("Delivd Msgs\t:", S_Msg_Out_Cnt)
    print("ACKd Msgs\t:", S_Msg_Delivery_Acked_Cnt)
    print("Full mboxes\t:", S_Mbx_Full_Cnt)
    print("Bad Cmd\t:", S_Bad_Req_Cnt)
    print("Bad Auth\t:", S_Bad_Auth_Cnt)
    print("Total Tx\t:", S_Tx_Cnt)
    print("User misses\t:", S_Bad_User_Cnt)
}
```

3.7 The Client Details

All messages sent to server by the client does so via the following function. `Cmd` is checked along with valid arguments. If something doesn't look right, then we exit the applicaiton after complaining.

```
function client_request(cmd, arg,    recipient, sender, msg, res, arr) {
    if (cmd == "send") {
        recipient = arg[0]
        sender = arg[1]
        msg = arg[2]
        if (msg == "") {
            print "Missing a message...\n  ...send <mbox> <from> <message>" \
                >"/dev/stderr"
            exit 1
        }
        gsub(/\t/, " ", msg)   # tabs are not allowed in message!
        if (length(msg) > Max_Message_Size) {
            print "Message too long, must be less than", Max_Message_Size, "bytes!" \
                >"/dev/stderr"
            exit 1
        }
        print send_request(compose_send_request(recipient, sender, msg))
```

```
    } else if (cmd == "check") {
        recipient = arg[0]
        if (fetch_msg(recipient, 0) != "") {
            print "You have messages!"
        } else {
            print "No Messages."
        }
    } else if (cmd == "listen") {
        recipient = arg[0]
        print "Polling every 2 seconds... Ctrl C to interrupt"
        while (1) {
            res = fetch_msg(recipient, 1)
            if (res != "") {
                print(res)
            } else  system("sleep 2")
        }
    } else if (cmd == "ping") {
        print send_request("PING")
    } else {
        print "Usage:" > "/dev/null"
        print " ... send <mbox> <from> <message>" >"/dev/stderr"
        print " ... check <mbox>" >"/dev/stderr"
        print " ... listen <mbox>" >"/dev/stderr"
        exit 1
    }
    exit 0
}
```

Check for a message, and optionally ACK it (so it can be deleted on the server side). Return the message or otherwise “”.

```
function fetch_msg(recipient, ack,  res, arr) {
    res = send_request(compose_check_request(recipient))
    split(res, arr, "\t")
    if (arr[1] == "OK") {
        msg = sprintf("%s\t%s\t%s\n", \
                strftime("%a %b %e %H:%M:%S %Z %Y", arr[3]+0), arr[2], arr[4])
        if (ack) send_request(compose_ack_msg_request(recipient))
        return msg
    }
    return ""
}
```

Here are the functions that a client uses to create requests. We haven't talked about the messaging protocols yet. You'll want to look at Messaging Protocol (§ 3.8.2) to get a grip on what these functions are composing.

```
function compose_send_request(recipient, sender, msg) {
    return sprintf("SEND\t%s\t%s\t%s", recipient, sender, msg)
}

function compose_check_request(name) {
    return sprintf("FETCH\t%s", name)
}

function compose_ack_msg_request(name) {
    return sprintf("ACK_MSG\t%s", name)
}
```

This function does the heavy lifting of encrypting and sending the message as well as decrypting and returning the response. If a response isn't heard form (e.g. the UDP packet is dropped), the client will hang here for up to 2 seconds before timing out. The UDP socket is closed and a new socket is created next time.

```
function send_request(req,     iv_s, iv, ec, mac, pt) {
    ec =encrypt(req, Auth_Key, IV)
    mac = siphash(req, Auth_Key_Bin, 16)
    iv_s = sprintf("%032x", IV)

    if (Debug) print("Encrypted msg=", ec, "mac=", mac, "iv=", iv_s)
    print(ec, mac, iv_s) |& Host_Addr;
    incr_IV_counter(int(length(ec)/16+1))

    PROCINFO[Host_Addr, "READ_TIMEOUT"] = 2000  # Sets a 2 second timeout
    if ( (Host_Addr |& getline) > 0) {
        iv = strtonum("0x" $3)
        if (Debug) print("Encrypted resp=", $1, "mac=", $2, "iv=", $3)
        pt = decrypt($1, Auth_Key, iv)
        hash = siphash(pt, Auth_Key_Bin, 16)
        if (hash !~ $2) {
            print "Bad response" >"/dev/stderr"
            return ""
        } else {
            return pt
        }
    } else {
        #  print "Timeout!" >"/dev/stderr"
        close(Host_Addr)
        return ""
    }
}
```

3.8 Messaging Overview

Now we finally get to the heart of the matter: Messaging. Message holds on the server are called mailboxes (yeah, clever, right?).

This is a poll based messaging system. Clients (Users) can do three things:

1. Send messages to the server to be stored for a named recipient in a mailbox.
2. Fetch messages that have been stored int named mailbox and receive them.
3. Delete the last fetched message from the mailbox.

That's it. There are a handful of guard rails, but this is not meant to be a comprehensive and feature rich messaging system. The primary focus is on being able to deliver messages securely to those who are authenticated/authorized users.

There is an option to set a lmaster *white list* of registered users, but there isis no way to check to see if a user has read their messages or even contacted the server. The white list is just to catch recipient mispellings, not for security. If someone polls your mailbox, then they get your messages.

A special "user" has been set up, named *all*. Sending to *all* will deliver the message to every mailbox on the server.

Oh, and since this is UDP, there is no guarantee of delivery.

That said, there are are a few protections:

- There is a limit to the size of messages that are stored
- There is a limit to the number of messages stored for any given 'name'. Mailboxes are organized as circular queue.

- An "OK" is sent by server to sender of message when it is successfully received.
- An "ERROR" is sent to the sender if the recipients "mailbox" is full or the recipient is not a white listed user.
- Messages aren't deleted from the mailbox until a *ACK_MSG* is sent or the mailbox overflows and the queue slot is overwritten by a new message.

The responses help mitigate the unreliability of UDP. See Messaging Protocol (§ 3.8.2) for a full decomposition of the protocol between client and server.

3.8.1 Mailboxes and Messages

Every message received by the server is stored in a recipient mailbox with the following meta-data:

- Date/Time received (as seconds since epoch)
- Sender name

A message can contain any printable character except a tab. Since we keep everything in neatly awk-able text format, A tab is reserved as a separator.

Mailboxes are associative arrays keyed with `recipient,slot`, where slot is a position in the mailbox queue. Each mbox has a top (the current first in queue) and a bottom (the next open slot in the queue). Recipients and senders are case insensitive (as a convenience for the sender).

As an example, *bob* could have a mailbox of two message that looks like this:

```
mbox["BOB",1] = "SUE\t1770476272\tHey Bob, this Susan"
mbox["BOB",2] = "NANCY\t1770476332\tJust reaching out. How are you?"
```

In the above example, Bob has 2 messages, one from SUE and the other from NANCY. They arrived about a minute apart. As, mentioned in Configurable Things (§ 3.1) Bob's mailbox can hold at most `Max_Mbox_Messages` number of messages. After this, sent message attempts to that mailbox are denied (with a NAK back to the sender).

The current message is kept track of by a mbox queue index `mbox_top[`*`BOB`*`]`, which would be `1` in the above case and the next slot would be `3` (`mbox_bot[`*`BOB`*`]`)

This would be the message sent when polled by BOB. Once BOB deletes the message then `mbox[`*`BOB`*`,top]` would increment to `2` and the contents of `mbox[`*`BOB`*`,1]` is freed.

This server function attempts to push a message into a mailbox.

```
function push(recipient, sender, msg,   dt, bot, top) {
    dt = systime()
    msg = substr(msg, 1, Max_Message_Size)
    if (White_List_String && (!(recipient in White_List) || !(sender in White_List))) {
        print("Unrecognized User!")
        S_Bad_User_Cnt++
        return 0
    }
    if (!(recipient in mbox_top)) {
        S_Mbx_Cnt++
        mbox_top[recipient] = 0
        mbox_bot[recipient] = 1
    } else {
        if (mbox_top[recipient] == mbox_bot[recipient])  return 0
    }
    top = mbox_top[recipient]
    bot = mbox_bot[recipient]
    mbox[recipient,bot] = sprintf("%s\t%d\t%s", sender, dt, msg)
    mbox_bot[recipient] = (bot == Max_Mbox_Messages) ? 1 : bot+1
    if (top == 0) mbox_top[recipient] = 1
    return 1
}
```

This server function attempts to fetch the top message in a mailbox without deleting (popping) it from the queue.

```
function fetch(recipient,   top) {
    top = mbox_top[recipient]
    if (top == 0) return ""
    return mbox[recipient,top]
}
```

Pop/delete a message from a mailbox.

```
function pop(recipient,    top, msg) {
    top = mbox_top[recipient]
    if (top == 0) return ""
    msg = mbox[recipient,top]
    delete mbox[recipient,top]
    top += 1
    if (top > Max_Mbox_Messages) top = 1
    mbox_top[recipient] = top
    if (top == mbox_bot[recipient]) {
        mbox_top[recipient] = 0
        mbox_bot[recipient] = 1
    }
    return msg
}
```

3.8.2 Messaging Protocol

The `message_processor` handles the full message protocol. All messaging is initated by the client and is completed by the server. All fields in the protocol are delimited by tabs.

SEND Send a message.

- Client: SEND *mbox sender message*
- Server: OK or ERROR

FETCH Fetch a message.

- Client: FETCH *mbox*
- Server: OK *message* or EMPTY

ACK_MSG Delete fetched message.

- Client: ACK_MSG "mbox"
- Server: OK or EMPTY

PING A test command to cause local stats to be printed. Also can be used by a client to test server liveliness and to get the initial IV it should use.

- Client: PING
- Server: PONG

You'll want to limit the number of messages stored per user via `Max_Mbox_Messages` as well as limited the maximum size of a message that can be sent. This should be smaller than the size of the safe UDP maximum size (508 bytes). Given a 32 byte hex encoded MAC and a 32 byte hex encoded IV/CTR separated by tabs, we can determine the maximum message size that can be encoded as base64.

The maximum message size is 330 bytes.

```
BEGIN {
    Max_Message_Size = int(3 *(508 - 32 - 1 - 32 - 1) / 4)-1
    if (Debug) print("Max Message Size =", Max_Message_Size)
}
```

This function runs our mailbox engine and returns stuff back to the user. This is the response side of the protocol. There is a special user "all" that delivers to everyone's mailbox.

```
function message_processor(buf,   ar, cmd, recipient, sender, msg) {
    split(buf, ar, "\t")
    cmd = ar[1]
    recipient = toupper(ar[2])
    if (cmd == "SEND") {
        S_Msg_In_Cnt++
        sender = toupper(ar[3])
        msg = ar[4]
        if (recipient == "ALL") {
            spam_whitelist(sender, msg)
            return "OK"
        } else if (push(recipient, sender, msg)) {
            return "OK"
        } else {
            S_Mbx_Full_Cnt++
            return "ERROR"
        }
    } else if (cmd == "FETCH") {
        msg = fetch(recipient)
        if (msg == "") return "EMPTY"
        S_Msg_Out_Cnt++
        return sprintf("OK\t%s", msg)
    } else if (cmd == "ACK_MSG") {
        if (pop(recipient) != "") {
            S_Msg_Delivery_Acked_Cnt++
            return "OK"
        } else return "EMPTY"
    }else if (cmd == "PING") {
        dump_stats()
        return "PONG"
    } else {
        S_Bad_Request_Cnt++
        return "HUH?"
    }

}
```

Sending messages to *everyone* (ALL) is a special case. We don't really track whether or not an individual delivery is successful or not. Instead we send to everyone who is in the white list. This function is only useful if you set up a white list. It does not spam mailboxes otherwise.

```
function spam_whitelist(sender, msg) {
    if (White_List_String) {
        for (recipient in White_List) {
            push(recipient, sender, msg)
        }
        return 1
    }
    return 0
}
```

3.9 Utilities and Back Matter

Here lies all the boring stuff...

3.9.1 Convert Hex encoded string to a byte string

This is horribly inefficient, but we aren't going for efficiency here...

```
function hex_to_bytes( hstr,     bstr) {
    for (i = 1; i <= length(hstr); i+= 2) {
        bstr = bstr sprintf("%c", strtonum("0x" substr (hstr, i, 2)))
    }
    return bstr
}
```

3.9.2 Convert a byte string to Hex encoded string

First we need an ordinal table to make this conversion a little faster.

```
function ord_init(    i, t) {
    for (i = 0; i <= 255; i++) {
        t = sprintf("%c", i)
        _ord[t] = i
    }
}
```

Now we do a more efficient conversion (compared to the inverse function)

```
function bytes_to_hex(bstr,    hstr) {
    for (i=1; i <= length(bstr); i++) {
        hstr = hstr sprintf("%02x", _ord[substr(bstr, i, 1)])
    }
    return hstr
}
```

Initialize our ordinal table.

```
BEGIN { ord_init() }
```

3.10 Crypto Functions

Here lies some pretty dense and gnarly code. For this to work, our work needs to be done in binary (numbers and byte strings) and not hex encoded strings. So, let's convert our auth key to binary.

Since I'm not sure that openssl will including a leading zero to keep the output, let's check anyway. I really should figure out, but for now let's just be safe...

```
BEGIN {
    if (length(Auth_Key) % 2)  Auth_Key = "0" Auth_Key;
    Auth_Key_Bin = hex_to_bytes(Auth_Key)
}
```

Now we can get on with the crypto algorithms.

3.10.1 AES-CTR

Ideally we would have a native awk implementation, but for now we will try and safely use openssl for now.

```
function encrypt (str, key, iv,  iv_s, cmd, estr) {
    iv_s = sprintf("%032x", iv)
    cmd = "openssl enc -aes-256-ctr -base64 -A -K " key " -iv " iv_s
    print str |& cmd
    close(cmd, "to")
    cmd |& getline estr
    close(cmd)
    return estr
}

function decrypt (estr, key, iv,    iv_s, cmd, str) {
    iv_s = sprintf("%032x", iv)
    cmd = "openssl enc -aes-256-ctr -d -base64 -A  -K " key " -iv " iv_s
    print estr |& cmd
    close(cmd, "to")
    cmd |& getline str
    close(cmd)
    return str
}
```

3.10.2 SipHash

The following is a pretty straightforward translation of the C reference implementation. This implementation uses the arbitrary precision feature of gawk (-M). I won't comment much further.

```
BEGIN {
    cROUNDS = 2
    dROUNDS = 4
}

function ROTL(x, b) { return and(0xFFFFFFFFFFFFFFFF,
                          or (lshift (x, b), rshift (x, (64-b)))) }
function ord(c) {
    return _ord[c]
}

function U8TO64_LE(str) {
    return or (ord(substr(str,1,1)), lshift(ord(substr(str,2,1)), 8), \
            lshift(ord(substr(str,3,1)), 16), lshift(ord(substr(str,4,1)), 24), \
            lshift(ord(substr(str,5,1)), 32), lshift(ord(substr(str,6,1)), 40), \
            lshift(ord(substr(str,7,1)), 48), lshift(ord(substr(str,8,1)), 56))
}

function U64TO8_LE(num) {
    num = and(num, 0xFFFFFFFFFFFFFFFF)
    return sprintf("%02x%02x%02x%02x%02x%02x%02x%02x", \
        and(num, 255), and(rshift(num, 8),255), \
         and(rshift(num, 16),255),  and(rshift(num, 24),255), \
         and(rshift(num, 32),255),  and(rshift(num, 40),255), \
         and(rshift(num, 48),255),  and(rshift(num, 56),255))
}

function SIPROUND() {
    v0 = and(v0+v1, 0xFFFFFFFFFFFFFFFF)
    v1 = ROTL(v1, 13)
```

```
    v1 = xor(v1, v0)
    v0 = ROTL(v0, 32)
    v2 = and(v2+v3, 0xFFFFFFFFFFFFFFFF)
    v3 = ROTL(v3, 16)
    v3 = xor (v3, v2)
    v0 = and(v0+v3, 0xFFFFFFFFFFFFFFFF)
    v3 = ROTL(v3, 21)
    v3 = xor(v3, v0)
    v2 = and(v2+v1, 0xFFFFFFFFFFFFFFFF)
    v1 = ROTL(v1, 17)
    v1 = xor(v1, v2)
    v2 = ROTL(v2, 32)
}

function siphash(inbuf, k,outlen,    inlen, b,m, left, outbuf) {

    v0 = 0x736f6d6570736575
    v1 = 0x646f72616e646f6d
    v2 = 0x6c7967656e657261
    v3 = 0x7465646279746573
    inlen = length(inbuf)
    left = and(inlen, 7)
    b = lshift(inlen, 56)
    k0 = U8TO64_LE(substr(k, 1, 8))
    k1 = U8TO64_LE(substr(k, 9))
    v3 = xor(v3,  k1)
    v2 = xor(v2, k0)
    v1 =xor(v1,  k1)
    v0 = xor(v0, k0)

    if (outlen != 8 && outlen != 16)  outlen = 16

    if (outlen == 16) v1 = xor(v1, 0xee)

    for (ni=1; ni < inlen; ni+=8) {
        m = U8TO64_LE(substr(inbuf, ni, 8))
        v3 = xor(v3, m)
        for (i=0; i < cROUNDS; ++i)  SIPROUND()
        v0 = xor(v0, m)
    }
    if (left >= 1) b = or(b, ord(substr(inbuf, ni, 1)))
    if (left >= 2) b = or(b, lshift(ord(substr(inbuf, ni+1, 1)), 8))
    if (left >= 3) b = or(b, lshift(ord(substr(inbuf, ni+2, 1)), 16))
    if (left >= 4) b = or(b, lshift(ord(substr(inbuf, ni+3, 1)), 24))
    if (left >= 5) b = or(b, lshift(ord(substr(inbuf, ni+4, 1)), 32))
    if (left >= 6) b = or(b, lshift(ord(substr(inbuf, ni+5, 1)), 40))
    if (left >= 7) b = or(b, lshift(ord(substr(inbuf, ni+6, 1)), 48))
    v3 =xor(v3, b)

    for (i=0; i < cROUNDS; ++i)  SIPROUND()
    v0 = xor(v0, b)
    v2 = xor(v2,  (outlen == 16) ? 0xee : 0xff)

    for (i=0; i < dROUNDS; i++)  SIPROUND()

    b =  xor(v0, v1,  v2, v3)
    out = U64TO8_LE(b)
    if (outlen == 8) return 0

    v1 = xor(v1, 0xdd)
```

```
    for (i=0; i < dROUNDS; i++)  SIPROUND()

    b = xor(v0, v1, v2, v3)
    out = out "" U64TO8_LE(b)
    return out
}
```

3.11 End Matter

This is the last BEGIN. It jumps to END (§ 3.5), which starts the application itself.

```
BEGIN {
    exit 0
}
```

3.12 Caveat emptor

Let the buyer beware!

This is NOT meant to be a system to be used for critical and secure communication. It's security has not been vetted. It was developed to entertain myself and perhaps educate. Squawk was developed to show that there doesn't have to be a lot of hand waving magic to implement something that just lets a handful of people communicate over the Internet. The only *black box* is the openssl (psuedo) random number genreation and implementation of AES-256-CTR. All else is here, right in the open.

The counter (CTR) approach (randomized seed from a 2^128 space) is highly suspect. There could be overlap and thus counter reuse (which is bad, bad, bad).

But, for less than 500 lines of Awk, what do you expect?

4 AWKZine

AWKZine? Is this a Zine? Well, in my very loose interpretation: Yes. I want to show you how to do stuff... in Awk. It's fringe. It's fun.

This was Issue #1. Hope you enjoyed it!

5 Revisions

version 1.5 - Unleashed upon unsuspecting public.

version 1.51 - Some typos fixed in the text.