A SLACK CLONE IN 5 LINES OF BASH

The title oversells the content a bit:

- first, Slack (or Mattermost, or even the Internet Relay Chat (IRC)) offer slightly more features than the *Simple Unix Chat* system (`suc`), the topic of this piece;
- then, `suc`'s actual line count exceeds five.

Nevertheless, `suc`'s core indeed consists of five lines of bash; and `suc` provides Slack, Mattermost, *etc.*'s core features:

- Real-time, rich-text chat,
- File sharing,
- Fine-grained access control,
- Straightforward automation and integration with other tools,
- Data encryption in transit
- and optionally at rest,
- state-of-the-art user authentication.

This paper shows how `suc` implements those features. `suc` stays small by leveraging the consistent and composable primitives offered by modern UNIX implementations [1].

Line count matters

> *One of my most productive days was throwing away 1000 lines of code. – Ken Thompson, apparently*

> *Measuring programming progress by lines of code is like measuring aircraft building progress by weight. – Bill Gates, (probably apocryphal)*

> *Some of the managers decided that it would be a good idea to track the progress of each individual engineer in terms of the amount of code that they wrote from week to week. […] When he got to the lines of code part, [Bill Atkinson] […] wrote in the number: -2000. – https://www.folklore.org/StoryView.py?story=Negative_2000_Lines_Of_Code.txt*

> *Their fundamental design flaws are completely hidden by their superficial design flaws. – Douglas Adams*

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. – Tony Hoare*

Despite the wide consensus among competent programmers that code is a liability, almost every widely-distributed piece of software is a complexity behemoth.

Case in point, let's examine Mattermost's line count:

```
cd /tmp
git clone --depth=1 https://github.com/mattermost/mattermost-server
cd mattermost-server
guix shell cloc -- cloc --quiet --timeout 0 .

github.com/AlDanial/cloc v 1.96  T=14.34 s (606.7 files/s, 139790.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Go                            1805          96705          26782         501249
JSON                           177              5              0         492604
TypeScript                    4125          74236          24557         480491
JavaScript                     811          21494          20745          68653
SCSS                           557           9164            359          51464
HTML                            54           6108           1167          37814
JSX                             92           3473           1054          29707
SQL                            807           3553           2253          18266
Text                            11           3824              0          10638
YAML                            45            126             96           7972
SVG                             68              6             12           2586
Markdown                        73            906             88           2168
make                             8            234             68            974
GraphQL                          4             65              2            596
XML                             29             10              1            572
Bourne Shell                    18            128             17            492
CSS                              5             70              0            385
Dockerfile                       4             14              8             46
CSV                              2              0              0             25
diff                             2              1             13              9
INI                              1              2              0              7
-------------------------------------------------------------------------------
SUM:                          8698         220124          77222        1706718
-------------------------------------------------------------------------------
```

Half a million lines of Go, and again **half a million** lines of TypeScript. Just for the server !

Let's compare with `suc`:

```
cd /tmp
git clone --depth=1  https://gitlab.com/edouardklein/suc
cd suc
guix shell cloc -- cloc --quiet --timeout 0 .

github.com/AlDanial/cloc v 1.96  T=0.01 s (475.8 files/s, 8207.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Bourne Again Shell               1              2              2             19
C                                1              3              3             17
make                             1              3              0             14
Bourne Shell                     1              0              1              5
```

```
--------------------------------------------------------------------------------
SUM:                            4             8             6            55
--------------------------------------------------------------------------------
```

`suc` can implement Mattermost's core features **with 0.005% of the code**. This is madness !

`suc`'s core loop

Behold the five lines of bash that do as much as half a million lines of Go:

```
while /usr/bin/true
do
    read -r line || exit 0  # EOF
    /usr/bin/echo "$(/usr/bin/date --iso-8601=seconds)"\
        "$(printf "%-9s" "$(/usr/bin/id --user --name --real)")" \
        "$line" >> /var/lib/suc/"$1"
done
```

This infinite loop:

- reads a line from standard input,
- prefixes it with:
    - the date,
    - the real user name,
- and appends it to a file in `/var/lib/suc/`

Surely, you think, this cannot do. What about authentication, access control, encryption, rich text, *etc.* ?

`suc` does all that by leveraging SSH, UNIX's access control API, and UNIX's text-based modularity.

Authentication

The `suc` process can only be launched by an authenticated user [2]. Therefore, `suc` contains no authentication code at all. All the authentication stuff happens before `suc` even starts.

As with almost all UNIX servers nowadays, remote authentication is handled by `ssh`. Before granting them the ability to start `suc`, `ssh` requires users to prove their identity.

This proof can take the form

- of a shared secret (*i.e.* a password),
- of a cryptographic challenge (as is the case on the dam),
- of the use of a One-Time-Passord (OTP) generating device,

- or of any combination of the above (also known as Multi-Factor Authentication, MFA).

`ssh` also authenticates the server to the client, thus preventing *Man-in-the-Middle* (MitM) attacks.

Last but not least, `ssh` encrypts all data between the clients and the server.

A successful installation of `suc` therefore depends on a correct configuration of the UNIX host and its `ssh` server. To use `suc`, a user needs to exist on the system; and the `ssh` server needs to be configured to let her remotely log in.

Most UNIX distribution provide the `useradd`, `passwd`, *etc.* commands for user management (creation, deletion, assignation to one or more groups, *etc.*). The `ssh` server reads its configuration from a text file in `/etc/` (typically `/etc/ssh/sshd_config`), and from public key files (typically in `/home/<user>/.ssh/authorized_keys`).

The dam server uses GNU Guix. GNU Guix differs from almost all other UNIX distributions, because it uses declarative configuration. This means that `root` just has to say what she wishes the configuration to be. The system then complies and reconfigures itself to match `root`'s declaration.

For example, granting `ssh` access to `alice` on a GNU Guix system [3] requires only the following line in the system's configuration file:

```
(ssh-user "alice" #:groups '("c3n" "frenchies")
                  #:keys '((plain-file "alice.pub" "SOMESSHKEY")))
```

User `alice` exists on the system only as long as the line exists in the configuration file. When the line disappears, the reconfiguration process removes user `alice`, and she can no longer log in.

Some big advantages of declarative configuration systems include:

- removing the need for clean up actions when removing functionality: once it is no longer part of the declaration, it will be removed from the system automatically.
- the ability to clone a specific configuration by just replicating the declaration; useful for back-ups, failovers, *etc.*.

Among the disadvantages, one counts an increased difficulty for quick and dirty setups (usually for a quick test to try out a piece of software). New tools (such as e.g. `guix shell`) allows one to sidestep this difficulty.

In such a declarative system, `suc`'s overhead per user is limited to a single line in the global configuration file. One cannot need less, and current chat systems need

more.

Access control

As with authentication, `suc` contains no access control code whatsoever. This combination of caring about neither authentication nor access control is called *security agnosticism*. *Security anosticism* allows `suc` to be lean, and therefore more probably correct (and so, paradoxically, more secure) than its heavier counterparts.

On UNIX, software can afford to be *security agnostic* because the system provides a clean and powerful API for access control: the kernel knows about

- users and groups,
- processes and files.

Let's dive in.

UNIX veterans will have noticed that `suc` prefixes the user's messages with her *real* name. Indeed, files have an owner (a user), whereas processes have two owners (two users). The *real* one and the *effective* one.

Most of the time, *real* and *effective* owners are the same. `suc`'s ownership differs: it *effectively* belongs to a special user also named `suc`; it *really* belongs to whoever (*e.g.* user `alice`) launched the `suc` command [4].

The kernel examines the *effective* ownership of a process to determine said process' ability to read or write to files.

With that in mind, let's examine the content of `/var/lib/suc` on the dam:

```
ssh -i ~/.ssh/id_rsa edk@the-dam.org ls -l /var/lib/suc

total 92
-rw-r----- 1 suc c3n            44368 Apr 13 19:18 banane
-rw-r----- 1 suc forbiddenlands  6234 Apr 13 21:04 forbiddenlands
-rw-r----- 1 suc frenchies         62 Apr 21 22:23 frenchies
-rw-r----- 1 suc guixdevs           0 Apr 22 15:39 guix
-rw-r----- 1 suc iwp9            4181 Apr 21 21:46 iwp9
-rw-r----- 1 suc users          18241 Jun 30 07:14 the-dam
-rw-r----- 1 suc wb3c             188 May 10 11:56 wb3c
```

The files in `/var/lib/suc` belong to `suc`; only `suc` can read and write those files [5].

Any other user, such as `alice`, may read some of the files (e.g. `banane`), provided she belongs to the appropriate *group* (e.g. `c3n`).

With this configuration, `suc` does not need to care about access control at all. For example `suc` need not match a user against the list of authorized readers or writers of a *channel*.

Instead, `usuc`[6] will just happily always *try* to read or write the file. The kernel will do the matching and prevent any unauthorized access.

On the dam, everyone can start `suc`, whose *effective* owner will be the user `suc`, who has the right to write into any channel. By design, any user on the dam can request membership into a group by blindly writing a request to the group's channel.

Less loosely-managed communities may wish to restrict channel write access to members only. `root` achieves this by maintaining multiple copies of the `suc` binary.

Let's assume that

- `alice` and `bob` belong to the `blue` group,
- while `eve` and `mallory` belong to the `red` group.

`root` creates `nobody`-like[7] users `red` and `blue`. She then creates two copies of `suc`, one for each group:

```
ls -l /usr/bin/suc*
total 32
-rwsr-xr-- 1 red     red      15624 Jun  4 10:51 suc_red
-rwsr-xr-- 1 blue    blue     15624 Jun  4 10:56 suc_blue
```

And she also creates one channel for each team:

```
ls -l /var/lib/suc/
total 16
-rw-r----- 1 blue     blue     11027 Jun  4 11:30 blue
-rw-r----- 1 red      red      17    Jun  4 10:53 red
```

One can see that:

- `alice` and `bob` belong to group `blue`.
  - They can read the `blue` channel. Indeed the file `/var/lib/suc/blue` belongs to group `blue` and has mode `-rw-r-----` : the second `r` means that members of the owning group (here, `blue`), can read the file (but not write to it).
  - They cannot directly write to the file. Only user `blue` can.
  - They can however launch the `/usr/bin/suc_blue` program, because group `blue` owns it, and it has mode `-rwsr-xr--` . The `x` means that members of the owning group (here, `blue`) can start the program.

    - This program will run with user `blue` as the *effective* owner: User `blue` owns the file and `root` has set its setuid bit (the `s` in the mode line says so).
    - Therefore, `alice` and `bob`, being members of the `blue` group, can launch the `/usr/bin/suc_blue` program, which being *effectively* owned by user `blue` (despite being launched by `alice` or `bob` who will be the *real*, but not *effective* owner) can write to the `/var/lib/suc/blue` file.
- `eve` and `mallory` belong to group `red` (but not group `blue`).
    - They cannot read the `blue` channel. Indeed, people other than user `blue` or members of group `blue` have no rights on the `/var/lib/suc/blue` file (the end of its mode line is `---`).
    - They cannot write to the `blue` channel directly, only user `blue` can.
    - They cannot start the `/usr/bin/suc_blue` program, because they do not belong to group `blue`. The only thing they can do to this file is read it (its mode line ends in `r--`).
    - Therefore they can neither read nor write the `blue` channel.

To relieve `root` from the cumbersome and error-prone process of setting this all up, `suc` provides an 80-something-lines long helper script called `suc_channel.sh`.

GNU Guix users can create a `suc` channel by adding a single line to the system's configuration file:

```
(suc-private-channel "red" "red")
```

This line takes care of creating the necessary

- `suc_red` setuid binary,
- `red` user
- `red` group
- `red` channel file.

Here, GNU Guix's declarative configuration paradigm shines again. The `suc_channel.sh` script may fail halfway, leaving the system in an undetermined state, whereas GNU Guix provides *transactional* updates: either the transition happens fully or it does not at all. The system always stays in a known clean state. One can even roll-back to a previous working state (see Multi-dimensional transactions and rollbacks, oh my!).

GNU Guix also automatically computes which groups, users, and setuid binaries should exist on the system. When `root` removes a private channel (e.g. `red`), she must assess whether the associated group (also named `red`), user (also `red`), and setuid binary (`suc_red`) should stay or go. That entails looking at the other

channels to see if any of them is still owned by user `red` or group `red`. Again, a cumbersome and error prone task whereas on GNU Guix, `root` just removes the channel's line from the system declaration. The `red` group, `red` user, and `suc_red` binary will stay if and only if another part of the system needs them.

As an illustration, here is a full system declaration for the above example. One can hardly be simpler than that.

```
(begin (use-modules
        (gnu packages base)
        (guix gexp)
        (beaver system)
        (beaver packages plan9)
        (beaver functional-services))
       (-> (minimal-ovh)
           (ssh-user "alice"   #:groups '("suc" "blue") #:keys '())
           (ssh-user "bob"     #:groups '("suc" "blue") #:keys '())
           (ssh-user "eve"     #:groups '("suc" "red")  #:keys '())
           (ssh-user "mallory" #:groups '("suc" "red")  #:keys '())
           (suc-private-channel "red" "red")
           (suc-private-channel "blue" "blue")
           (suc-public-channel "purple")))
```

Fancy text

We have seen how `suc` is *security-agnostic*, relying on:

- `ssh` for authentication,
- UNIX's file and process ownership and permission model for access control.

Let's now dive into the featureful side of things by first looking at some bells and whistles: rich text.

Most chat applications nowadays piggyback on an HTML engine to render the chat's text. For example mattermost's client uses Electron. There go another few tens of thousand of lines of code.

On the one hand, this adds tremendous complexity and increases the attack surface of the application. On the other hand it lets the chat display elements in a complex layout, or embed interactive widgets within the messages (such as emoji reactions), etc.

`suc` uses one file per channel. This text file is meant to be displayed to the user with a command-line tool such as `tail` or `cat`.

Before everything got shoehorned into an HTML rendering engine, people managed to display rich text, boxes, and even primitive graphics on their terminals. These capabilities more-or-less coalesced into something called ANSI escape codes[8]. Almost all terminal emulators support those. Together with proper UTF-8 support, they allow for the colorful, emoji-filled experience of your average corporate slack channel, with ~5% of the memory footprint.

If you paid attention to the 5 lines of bash that `suc` consists of, you have noticed that while `suc` writes into the channel file, it does not read from it.

This job befalls to `usuc`. Why two separate binaries ? Because `suc` is a privileged binary, which runs under the powerful effective ownership of whoever can write to a channel. One must be careful to keep the logic and external dependencies of `suc` to a bare minimum to minimize the attack surface, and avoid any complex logic where bugs like to hide.

`usuc`, conversely, runs with both effective and real owners set to the calling user. It can go crazy with the features, as whatever happens can not impact the channel file, except through `suc`, whose logic is so simple there should not be any bugs in it.

Here is as of <2023-06-29 Thu> the code for usuc:

```
#!/usr/bin/bash
set -euo pipefail

# Autowrap self in rlwrap
if [ -z "${RLWRAP:-}" ]
then
    RLWRAP=1 rlwrap "$0" "$@"
    exit 0
fi

chan_owner=$(ls -l /var/lib/suc/"$1" | cut -d' ' -f 3)
if [ "$chan_owner" != suc ]
then
    SUC=suc_"$chan_owner"
else
    SUC=suc
fi
# Tail the channel
tail -f -n 20 /var/lib/suc/"$1"&
while true
do
    read -r line || exit 0
    if [ "${line::1}" == ":" ]
    then
        echo '*runs* `' "${line:1}" '`' | pygmentize -l md -f 256 | "$SUC" "$1"
        bash -c "${line:1}" | "$SUC" "$1"
    else
        echo "$line" | pygmentize -l md -f 256 | "$SUC" "$1"
    fi
done
```

`usuc`:

- makes sure to prefix its own call with `rlwrap`, which provides history and line editing capabilities,
- selects the correct setuid `suc` binary to run depending on who owns the channel file,
- calls `tail -f`, displaying the last 20 lines of the channel and then anything that get subsequently written to it,
- check whether the line typed by the user starts with ":" (see the next section),

- pipe anything the user typed through `pygmentize`.

Pygmentize is a nifty Python module for syntax coloring. Here it runs expecting markdown on its standard input, and outputting ANSI color coded text on its standard output. That way, a user can use markup syntax like `**bold**`, and get **bold** output. `suc` gets markdown support in a single line of code.

Chat commands

Other tools can, like `pygmentize`, output ANSI-styled text. One of those is e.g. `gum`.

To invoke `gum` directly from the chat interface, one just has to start a message with `:`. `usuc` will catch that and will not pipe the text to `suc` like it would for a normal message. It will instead run the command, and pipe its *output* to `suc`.

One can therefore type:

```
: gum style --border=rounded --bold --foreground=#F00 "Hello World !"
```

as a `suc` message and see something that looks like the following appear in the channel:

Any command that exists in the namespace of the user who called `usuc` can run that way. Its output will appear in the chat.

We use that on the dam to roll dice when we play table-top role playing games:

```
: roll 2d6
2023-04-13T21:04:57+00:00 gm        *runs* ` roll 2d6 `
2023-04-13T21:04:58+00:00 gm        [6, 2]
```

Again, it all happens in the namespace of the user. Any user can customize her environment to keep useful chat macros on hand, without any impact on the other users.

Piping text to `suc`

Instead of using `usuc`'s command-calling facility, one can pipe right into `suc` the output of any command, from one's shell.

For example if you want to pretty-print a piece of source code to a relevant channel, you can invoke `bat`:

```
bat --force-colorization --paging=never --style=full toto.c | suc greybeards
```

and you will get a syntactically-colored listing of your code in the channel.

Complex chat system like Mattermost, Slack, etc. offer many integrations, that is, ways to interact with other software.

`suc` is text-based ; integrating it with other tools feels natural in a UNIX environment. For example consider the following bash one-liner:

```
make test > testlog || (suc devops < testlog ; exit 1)
```

This code will run the tests of a software project, and send the logs to the `devops` channel on failure.

With the necessary boilerplate, this oneliner fits into the git hook `update` of a git repo:

```
#!/usr/bin/bash
set -euxo pipefail
newrev="$3"

GIT_DIR=$(realpath "$GIT_DIR")
cd "$(mktemp -d)"
git clone "$GIT_DIR" .
git checkout "$newrev"

make test > test_log || (suc devops < test_log ;  exit 1)
exit 0
```

And voilà ! You get a `git/suc` integration in 11 lines of bash. Any push to the repo will trigger the test, reject the update on failure, and ring the DevOps team so they can solve the problem.

Reading from a `suc` channel

`suc` users continually update a text file (the channel). By calling `tail -f` on that text file, you can process the new lines as they arrive.

For example, to get notified when a new message gets posted in a channel, just run:

```
tail -n0 -f /var/lib/suc/some-chan | (while true;
                                      do read -r line;
                                         notify-send "$line";
                                      done)
```

Too many notifications ? Reduce the noise by grepping for keywords:

```
tail -n0 -f /var/lib/suc/some-chan | \
    stdbuf -i0 -o0 grep -E "(myname|build failure|fire)" | \
    (while true; do read -r line; notify-send "$line"; done)
```

Don't want to open as many windows as channels you follow ? Coalesce them all in a single feed:

```
tail -f /var/lib/suc/*
```

Or use the more powerful lnav (a log file viewer), which will

- remember where you left off,
- set bookmarks,
- assign a color to each channel,
- parse the date, username, or any custom field that may appear in the text,
- let you filter the messages,
- run SQL queries on the messages.

Try to do that with Slack…

Bots

If you can write and read to a `suc` channel, you can do both at once. Chat systems often host bots and semi-automated "assistants". These provide a text-based interface to e.g. tickets, continuous integration, corporate directory, server logs, etc. Have a look below at the code of a bot that convert into meters any length given in feet:

```
#!/usr/bin/bash
feet_to_meters (){
    feetexpr="$1"
    echo -e "$feetexpr \n m" | units | grep -Eo "\* [0-9.]*" | tr -d '*'
}

tail -n0 -f /var/lib/suc/"$1" | \
    stdbuf -i0 -o0 grep -v "metric_bot"  | \
    stdbuf -i0 -o0 grep -Eo "[0-9]+[[:blank:]]*(feet|ft)" | \
    (while true;
     do read -r line;
        echo "[metric_bot] $line is $(feet_to_meters "$line") meters." | suc "$1"
     done)
```

```
2023-06-30T11:20:47+02:00 edouard   The plane flew at 33000 ft.
2023-06-30T11:20:47+02:00 bots      [metric_bot] 33000 ft is  10058.4 meters.
```

Conclusion

`suc` piggybacks on SSH for authentication and on UNIX for access control and composability. It provides almost all the features offered by Mattermost, Slack, *etc.* with such a ridiculously small fraction of the code that one wonders why such complex systems even exist.

Using text files as the base for `suc` channels lets user leverage UNIX tools for reading (`tail`, `bat`, `lnav`, `less`, `grep`, etc.), writing (`gum`, `bat`, `pygmentize`, etc.), or semi-automated extension with bots, hooks, and scripts.

Tools can be written in any language, as long as they read and write text.

Advertisement

If you want to play with `suc` but don't want to bother with installing it, or if you don't have any friends to share a `suc` instance with, come and join us at the dam ! For a measly 10€/year, you can enjoy sharing `suc` on a GNU Guix server with people from all over the world.

If you would like your own instance of `suc`, don't hesitate and rent a VPS from Guix hosting ! For 100€/year, you get a GNU Guix VPS. Adding `suc` is just one line of configuration away. There are no usage-based restrictions, your data stays yours, and you can use your VPS to provide other services as well.

Footnotes:

1

in the dam case, GNU Guix, but `suc` itself contains no Guix- or even Linux-specific code

2

Usually done by calling `usuc <some-channel>` on the command line

3

configured with Beaver Labs' channel (see https://gitlab.com/edouardklein/guix)

4

UNIX exposes this capability through something called the `setuid` bit.

5

If you do not understand the `-rw-r-----` output of the command above, I recommend section 2.4, "Permissions" of Brian W Kernighan and Rob Pike, The UNIX Programming Environment, (Prentice-Hall Englewood Cliffs, NJ, 1984).

6

`usuc` extends `suc` with bells and whistles, see below.
[7]

`nobody` is usually a passwordless, shellless user who owns no files, to whom ownership of a process is transferred when one wants to prevent said process from being allowed to do any damage on the system.

[8]

This is a very deep rabbit hole. If you want to dive in, I've found good starting points to be: Nick Black's "Hacking the Planet (with Notcurses). A Guide to TUIs and Character Graphics", the relevant section of the VT100 manual, or this list of animations.