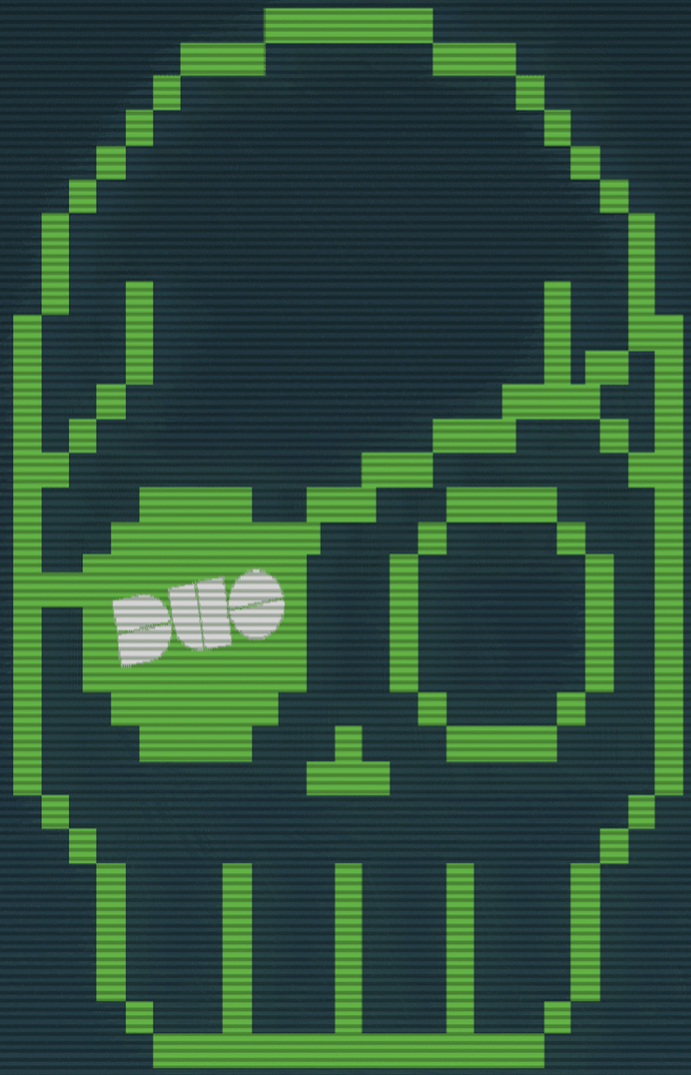
From Zero to Zero-Trust:
Lessons Learned Building a BeyondCorp SSH Proxy



James Barclay

@futureimperfect

Senior R&D Engineer

Duo Labs

Agenda

1. BeyondCorp 101
2. The Access Proxy
3. The Access Proxy and SSH
4. The Server
5. The Client
6. Questions

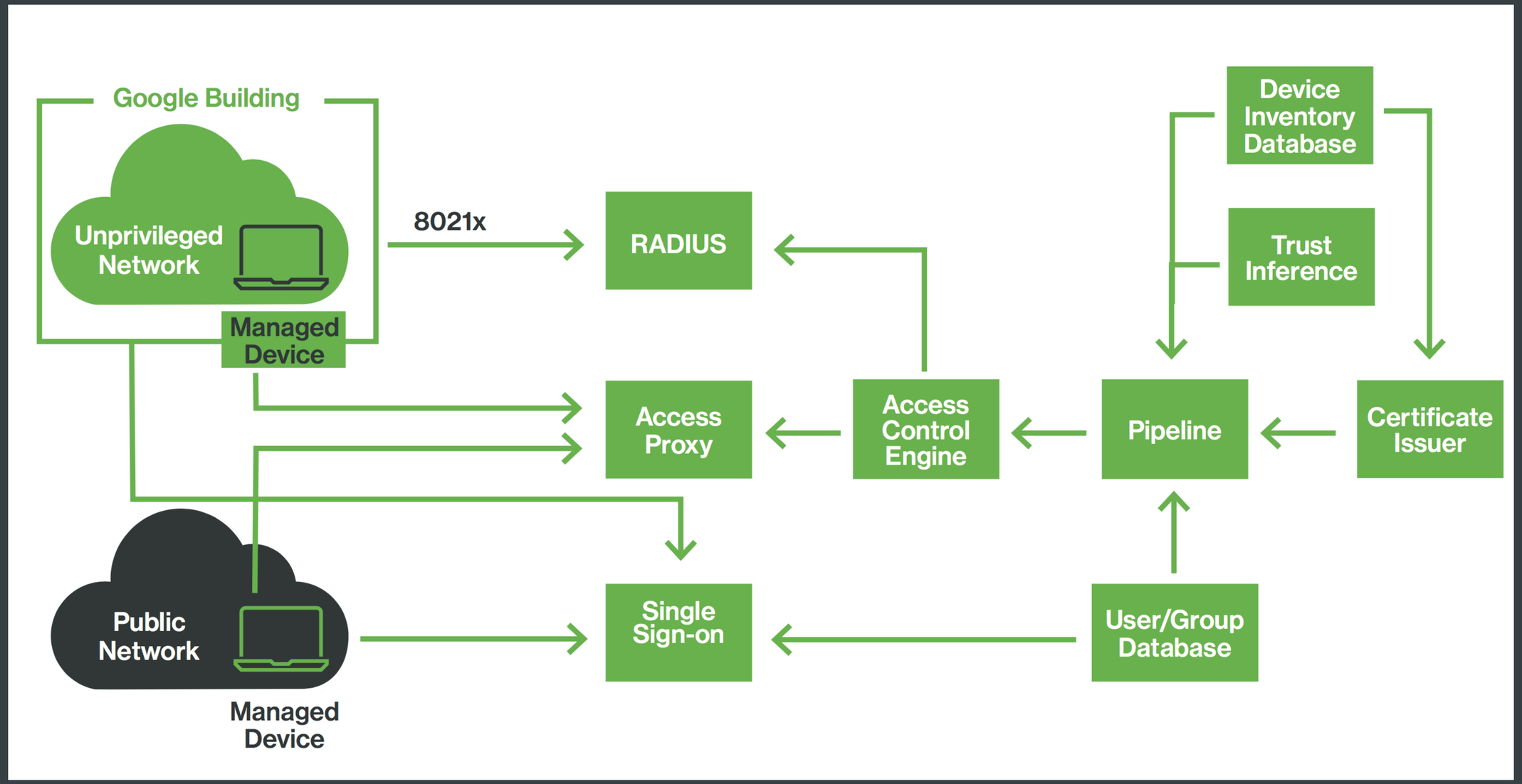
BeyondCorp 101

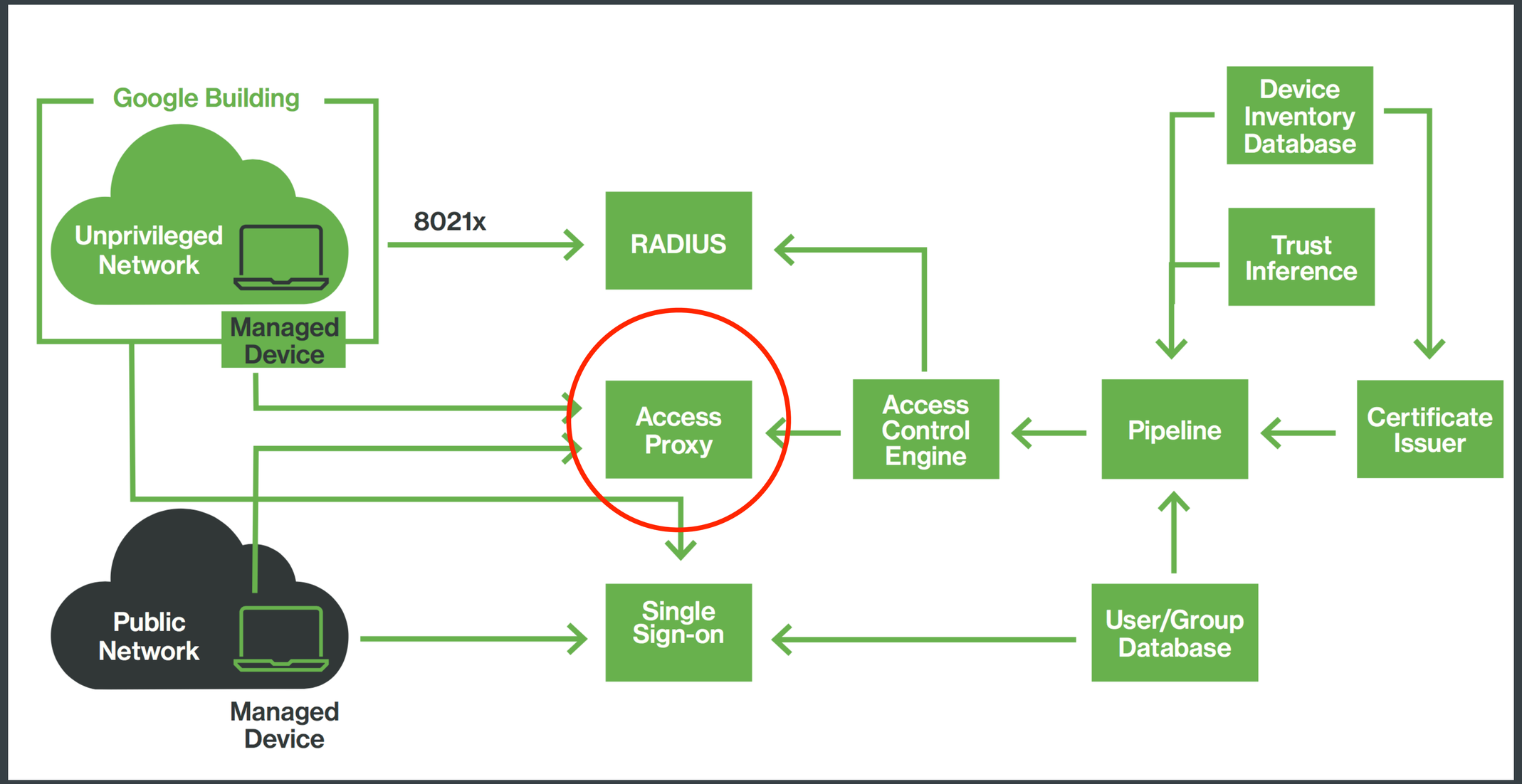

Data
Attackers

Replace Trust in the Network With Trust in the Device + User

BeyondCorp 101

- It doesn't matter if you're working from a ☕ shop or HQ, you get the same checks
- **Example 1:** To access the lunch menu, you must have a managed device
- **Example 2:** To access source code repos, you must have a managed device, the latest software updates, MFA, etc.


Google Building
Unprivileged Network
Managed Device
8021x
RADIUS
Device Inventory Database
Trust Inference
Access Proxy
Access Control Engine
Pipeline
Certificate Issuer
Public Network
Managed Device
Single Sign-on
User/Group Database


Google Building
Unprivileged Network
Managed Device
8021x
RADIUS
Device Inventory Database
Trust Inference
Access Proxy
Access Control Engine
Pipeline
Certificate Issuer
Public Network
Managed Device
Single Sign-on
User/Group Database

The Access Proxy

Terminology

- **Access Proxy:** A web server + reverse proxy responsible for authorizing and proxying requests, (HTTP/s, SSH, etc.)
- **Service:** A web/SSH/RDP/VNC/etc. server that sits behind the reverse proxy

Access Proxy Who?

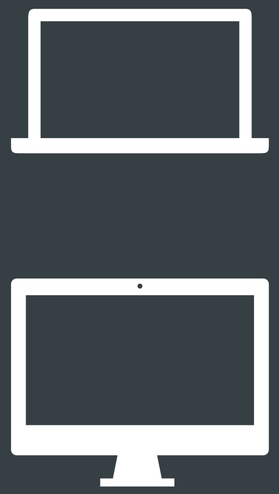
- A web app that handles determining whether a user is authorized to access a service
- A reverse proxy that communicates with the web app to determine if a request is authorized

Service What?

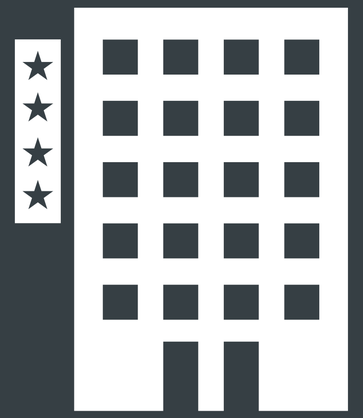
- A service, in practice, is a DNS name, (e.g., wiki.example.com or ssh.example.com)
- Each service has an external DNS name and internal hostname or IP
- Users are authorized to access services by the access proxy

The Access Proxy and Web Applications

- The most common reason to use our VPN internally was to access on-premise web applications
- We wanted a way for employees to securely access on-premise web applications without a VPN, so we solved for this first



Access
Proxy



External
Clients

Perimeter
Firewall

Internal
Firewall

On-Premise
Stuff

The Access Proxy Authentication Flow

- The Nginx `auth_request` directive handles authorization based on the result of a subrequest
- If the subrequest returns a `2XX` response code the request is authorized
- If a `401` or `403` is returned access is denied

The Access Proxy Authentication Flow, cont.

- The subrequest checks for a valid session cookie for the particular service
- For unauthorized requests, we initiate an SSO flow by returning a `401` and having Nginx redirect to the login URL
- Nginx proxies authorized requests to backend services

```
server {
    # Require authorization.
    auth_request /verify;

    # Redirect to the login handler if the access
    # proxy says the service session is invalid.
    auth_request_set $access_proxy_check $upstream_http_x_access_proxy_check;
    error_page 401 $access_proxy_check;

    location = /verify {
        internal;
        auth_request off;
        proxy_pass http://unix:@accessproxy;
        proxy_set_header Host $http_host;
        proxy_set_header X-Original-URI $request_uri;
        ...
    }
}
```

The Access Proxy and SSH

SSH is second only to HTTP(s) for our internal VPN usage

In order to set our VPN on 🔥 we needed a solution for SSH

1. **Must** be easy to add new services

2. **Must** work with existing SSH tooling, (OpenSSH, Chrome Secure Shell)

3. **Must** be able to use a browser-based authentication flow

```
SSH(1)                    BSD General Commands Manual                    SSH(1)

NAME
     ssh -- OpenSSH SSH client (remote login program)

SYNOPSIS
     ssh [-1246AaCfGgKkMNnqsTtVvXxYy] [-b bind_address] [-c cipher_spec] [-D [bind_address:]port] [-E log_file]
         [-e escape_char] [-F configfile] [-I pkcs11] [-i identity_file] [-J [user@]host[:port]] [-L address]
         [-l login_name] [-m mac_spec] [-O ctl_cmd] [-o option] [-p port] [-Q query_option] [-R address] [-S ctl_path]
         [-W host:port] [-w local_tun[:remote_tun]] [user@]hostname [command]

DESCRIPTION
     ssh (SSH client) is a program for logging into a remote machine and for executing commands on a remote machine.  It
     is intended to provide secure encrypted communications between two untrusted hosts over an insecure network.  X11
     connections, arbitrary TCP ports and UNIX-domain sockets can also be forwarded over the secure channel.

     ssh connects and logs into the specified hostname (with optional user name).  The user must prove his/her identity
     to the remote machine using one of several methods (see below).

     If command is specified, it is executed on the remote host instead of a login shell.

     The options are as follows:

     -1      Forces ssh to try protocol version 1 only.

     -2      Forces ssh to try protocol version 2 only.

     -4      Forces ssh to use IPv4 addresses only.

     -6      Forces ssh to use IPv6 addresses only.

     -A      Enables forwarding of the authentication agent connection.  This can also be specified on a per-host basis
             in a configuration file.

             Agent forwarding should be enabled with caution.  Users with the ability to bypass file permissions on the
             remote host (for the agent's UNIX-domain socket) can access the local agent through the forwarded connec-
             tion.  An attacker cannot obtain key material from the agent, however they can perform operations on the
             keys that enable them to authenticate using the identities loaded into the agent.

     -a      Disables forwarding of the authentication agent connection.

     -b bind_address
             Use bind_address on the local machine as the source address of the connection.  Only useful on systems with
             more than one address.

     -C      Requests compression of all data (including stdin, stdout, stderr, and data for forwarded X11, TCP and
             UNIX-domain connections).  The compression algorithm is the same used by gzip(1), and the ``level'' can be
             controlled by the CompressionLevel option for protocol version 1.  Compression is desirable on modem lines
             and other slow connections, but will only slow down things on fast networks.  The default value can be set
             on a host-by-host basis in the configuration files; see the Compression option.

     -c cipher_spec
             Selects the cipher specification for encrypting the session.

             Protocol version 1 allows specification of a single cipher.  The supported values are ``3des'',
             ``blowfish'', and ``des''.  For protocol version 2, cipher_spec is a comma-separated list of ciphers listed
             in order of preference.  See the Ciphers keyword in ssh_config(5) for more information.

     -D [bind_address:]port
             Specifies a local ``dynamic'' application-level port forwarding.  This works by allocating a socket to lis-
             ten to port on the local side, optionally bound to the specified bind_address.  Whenever a connection is
             made to this port, the connection is forwarded over the secure channel, and the application protocol is then
             used to determine where to connect to from the remote machine.  Currently the SOCKS4 and SOCKS5 protocols
             are supported, and ssh will act as a SOCKS server.  Only root can forward privileged ports.  Dynamic port
             forwardings can also be specified in the configuration file.
```

We also strongly preferred to keep the same authentication flow as we had for web apps behind the access proxy

Using a browser-based authentication flow made this possible

The access proxy should also be transparent to backend services, regardless of the underlying protocol

For non-HTTP protocols, WebSockets made this easy

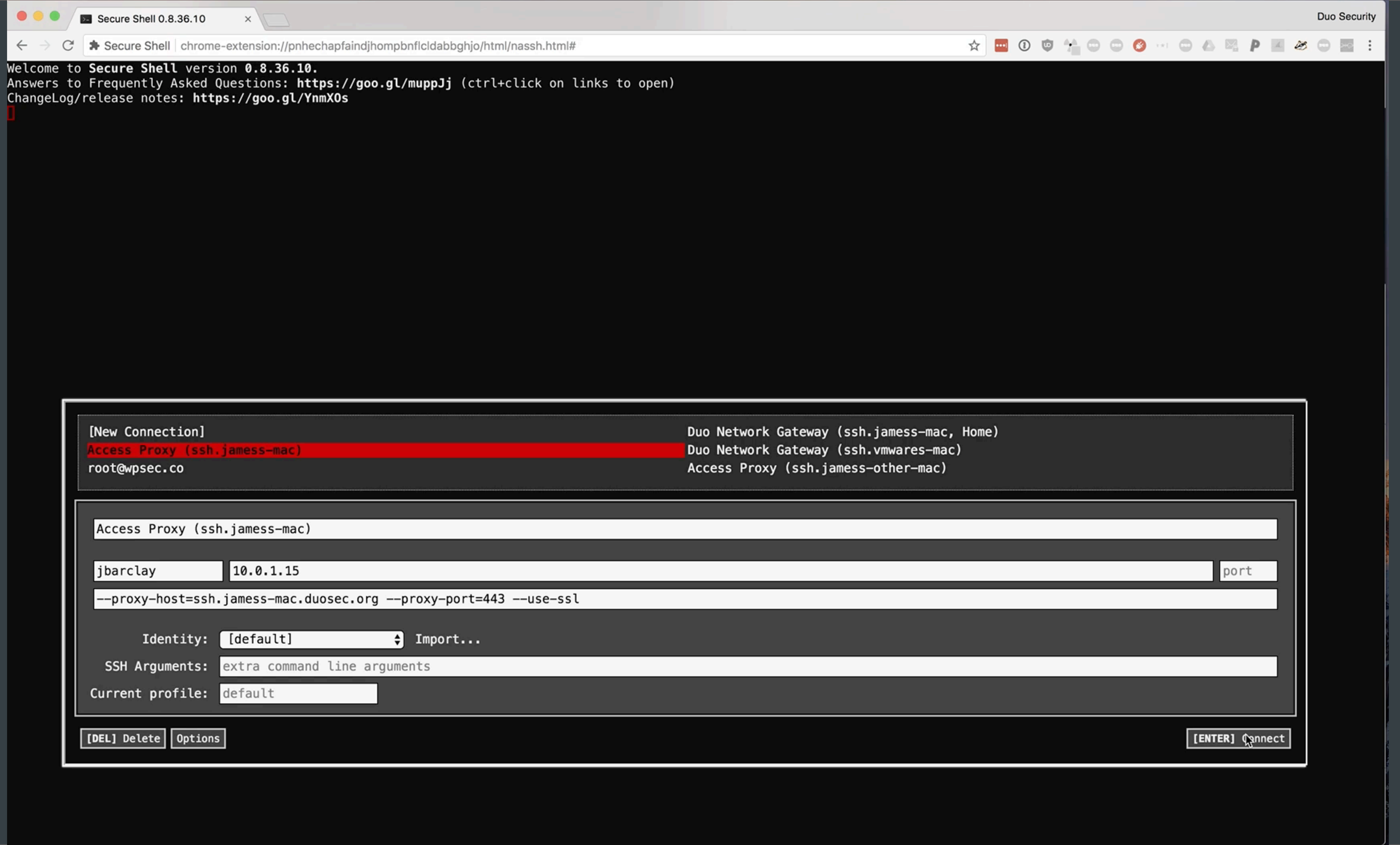
"Wrapping SSH traffic in HTTP over TLS is easy thanks to the existing `ProxyCommand` facility. We developed a local proxy which looks a lot like Corkscrew, except the bytes are wrapped into WebSockets..."

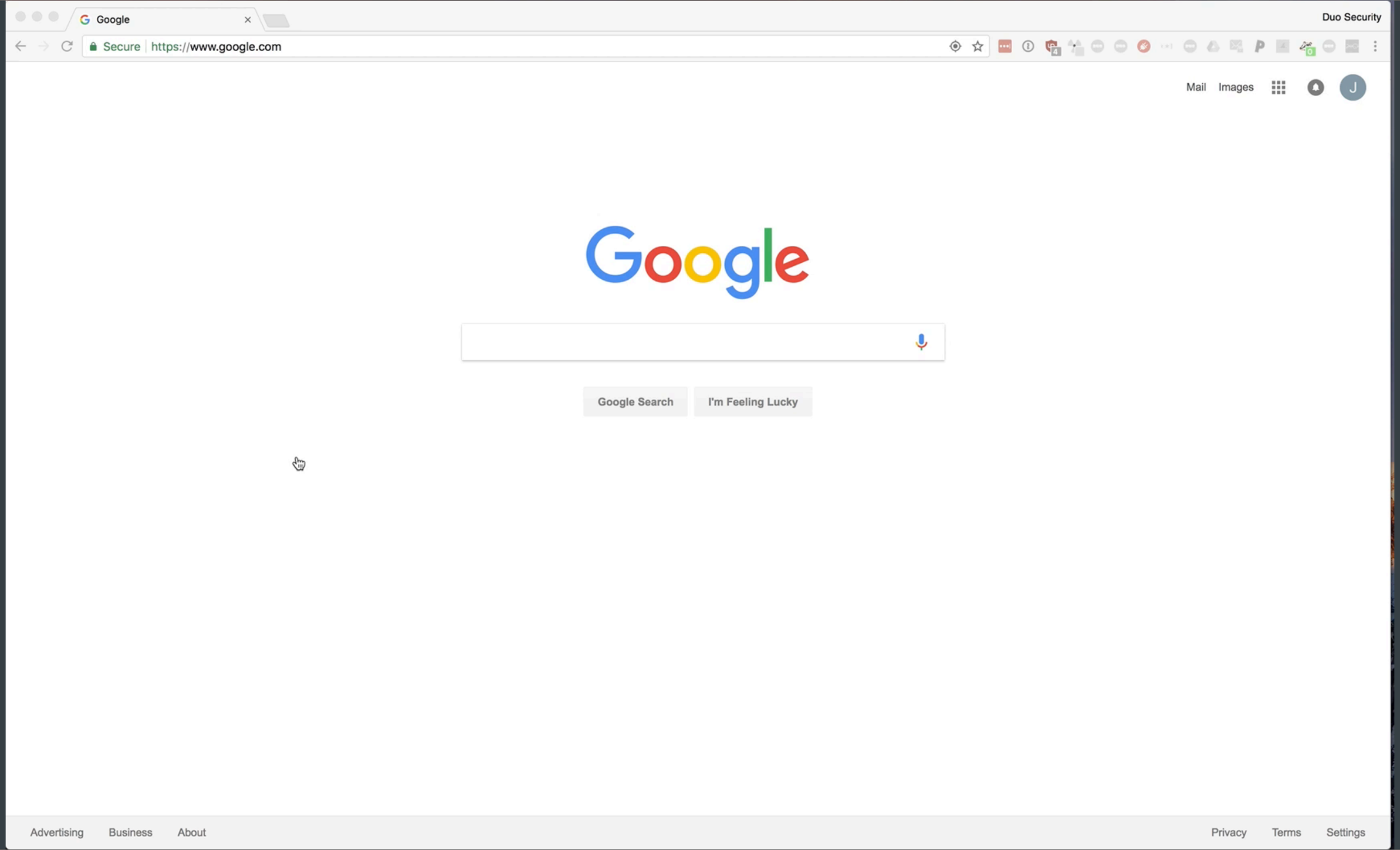
BeyondCorp Part III: The Access Proxy

nassh_google_relay.js

```
/**
 * This file contains the support required to make connections to Google's
 * HTTP-to-SSH relay.
 *
 ...
 *
 * The relay is only available within Google at the moment.  If you'd like
 * to create one of your own though, you could follow the same conventions
 * and have a client ready to go.
```


YAS!


Secure Shell 0.8.36.10
Duo Security
Secure Shell
chrome-extension://pnhechapfaindjhompbnflcldabbghjo/html/nassh.html#
Welcome to Secure Shell version 0.8.36.10.
Answers to Frequently Asked Questions: https://goo.gl/muppJj (ctrl+click on links to open)
ChangeLog/release notes: https://goo.gl/YnmXOs
[New Connection]
Access Proxy (ssh.jamess-mac)
root@wpsec.co
Duo Network Gateway (ssh.jamess-mac, Home)
Duo Network Gateway (ssh.vmwares-mac)
Access Proxy (ssh.jamess-other-mac)
Access Proxy (ssh.jamess-mac)
jbarclay
10.0.1.15
port
--proxy-host=ssh.jamess-mac.duosec.org --proxy-port=443 --use-ssl
Identity:
[default]
Import...
SSH Arguments:
extra command line arguments
Current profile:
default
[DEL] Delete
Options
[ENTER] Connect

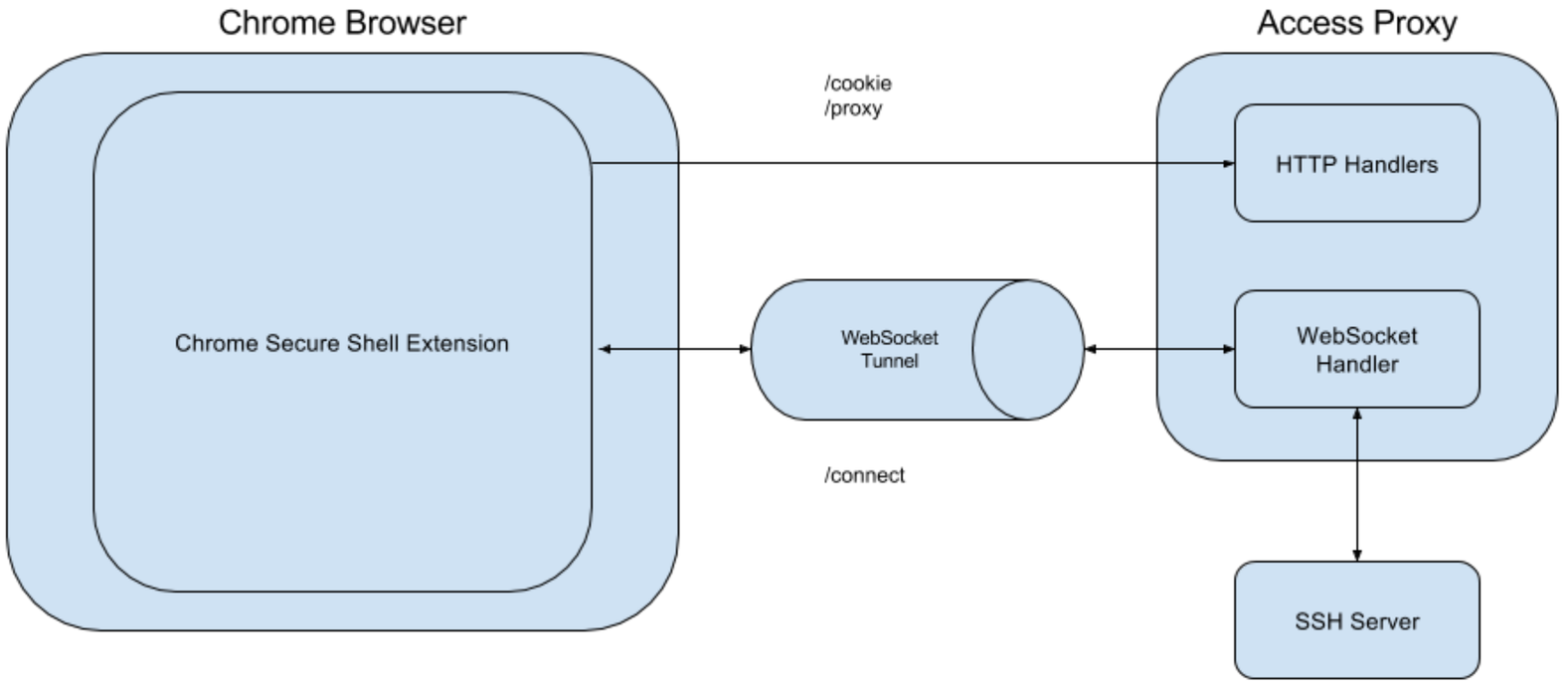

Google
Duo Security
Secure https://www.google.com
Mail
Images
J
Google
Google Search
I'm Feeling Lucky
Advertising
Business
About
Privacy
Terms
Settings

How It Works, In a Nutshell

- **Chrome Secure Shell:** Client > WebSockets > Access Proxy > Backend
- **OpenSSH:** Client > ProxyCommand > Local Proxy > WebSockets > Access Proxy > Backend

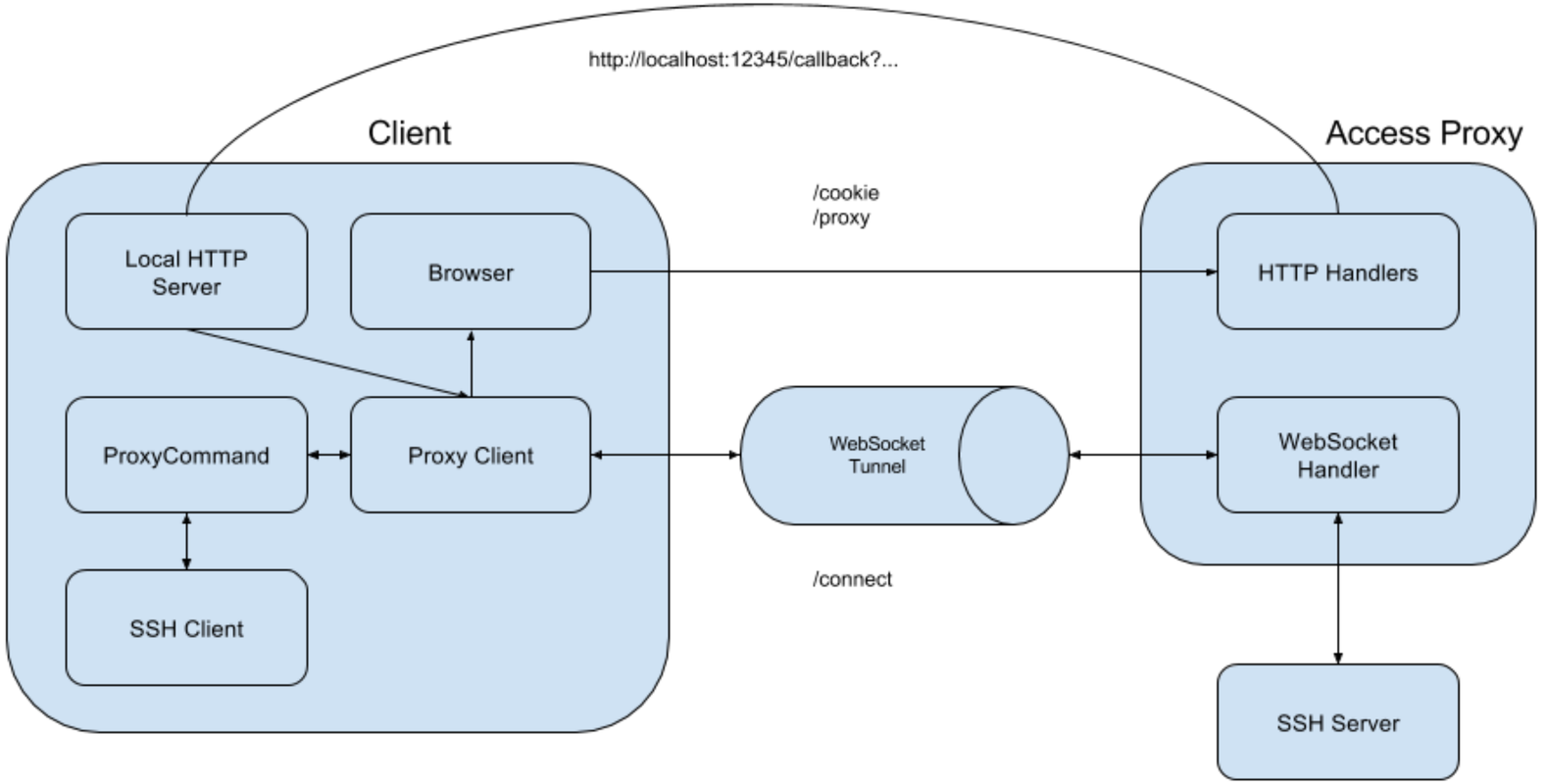
Example URLs

- https://ssh.example.com/cookie?ext=<CHROME_EXTENSION_ID>&path=/callback
- chrome-extension://<CHROME_EXTENSION_ID>/#user@ssh.example.com:443
- https://ssh.example.com/proxy?host=192.168.1.1&port=22
- https://ssh.example.com/connect?sid=<UUID>&ack=0&pos=0



Example URLs

- https://ssh.example.com/cookie?path=/callback
- http://localhost:12345/callback?relay=ssh.example.com&server=192.168.1.1&port=22&_COOKIE_=<COOKIE>
- https://ssh.example.com/proxy?host=192.168.1.1&port=22
- https://ssh.example.com/connect?sid=<UUID>&ack=0&pos=0



The NaSSH Relay Protocol

Chrome Secure Shell

- Chrome Secure Shell supports the NaSSH relay protocol, documented in `libapps-master-nassh/js/nassh_google_relay.js`
- Google doesn't provide source for the relay itself, but they describe it as an HTTP-to-SSH relay that uses WebSockets and the handlers are well documented

By following the NaSSH relay protocol our proxy worked with Chrome Secure Shell

This is cool because we ❤️ Chrome OS

WebSockets

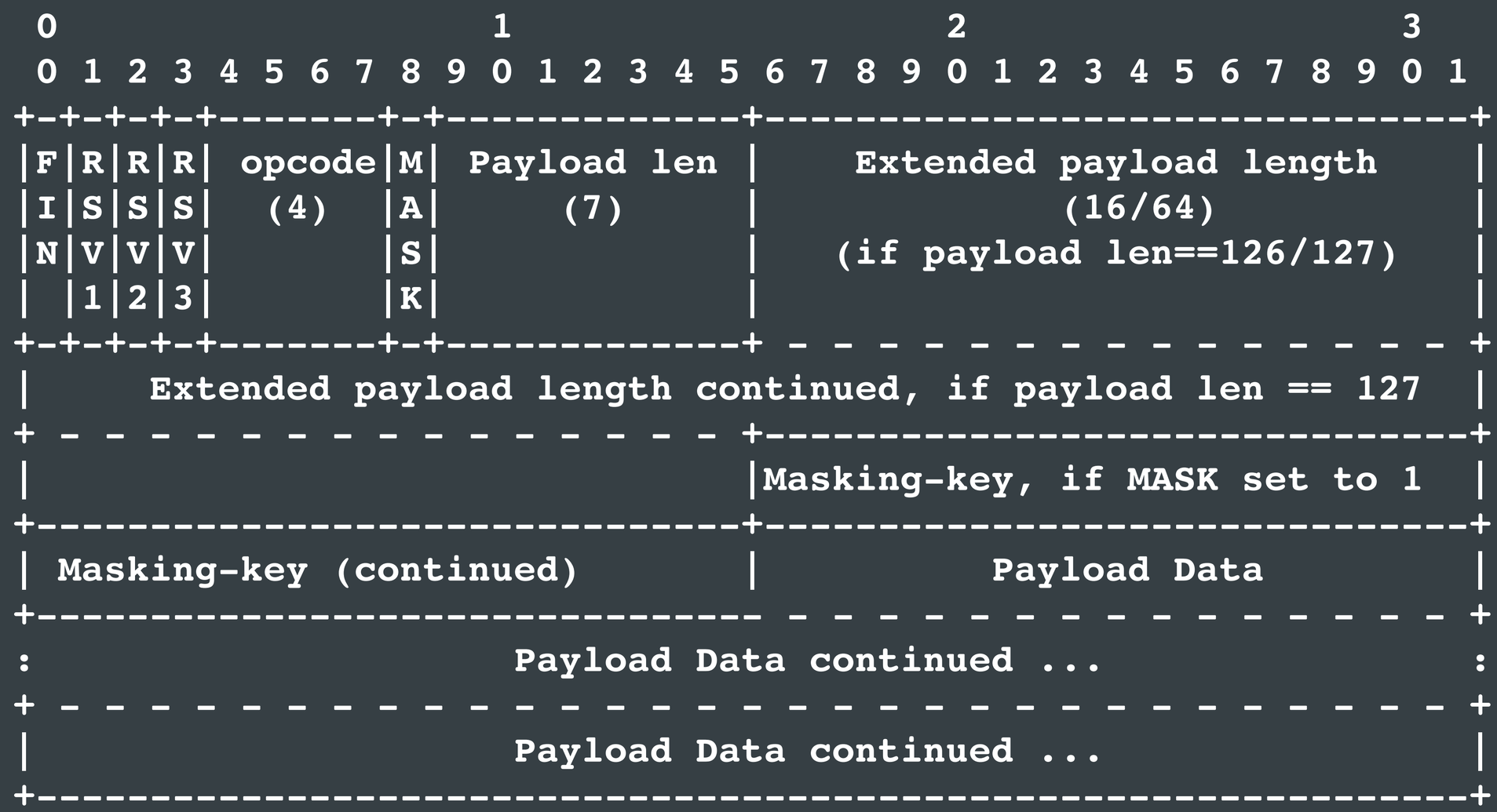
- Basic message framing layered over TCP (RFC 6455)
- Designed for two-way communication between browsers and servers, without opening multiple HTTP connections
- Allows us to separate device authentication from user authentication, (device cert for TLS handshake, `(password || SSH key)` for SSH authentication)

WebSockets, cont.

- The WebSocket client handshake is an HTTP upgrade request
- Once the connection is established, messages are passed over a persistent connection
- The connection can be closed by client or server by sending a close control frame

```
GET /connect HTTP/1.1
Host: ssh.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: https://example.com
Sec-WebSocket-Version: 13
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
FIN
RSV1
RSV2
RSV3
opcode (4)
MASK
Payload len (7)
Extended payload length (16/64) (if payload len==126/127)
Extended payload length continued, if payload len == 127
Masking-key, if MASK set to 1
Masking-key (continued)
Payload Data
Payload Data continued ...
Payload Data continued ...

NaSSH?

"Secure Shell (nassh) is a Chrome App that combines hterm with a NaCl build of OpenSSH to provide a PuTTY-like app for Chrome users"

nassh/README.md

NaCl?

"Native Client is a sandbox for running compiled C and C++ code in the browser efficiently and securely, independent of the user's operating system"

Welcome to Native Client

hterm?

"hterm is a JS library that provides a terminal emulator...Do not confuse this with an ssh client (like Secure Shell) or a shell environment by itself. It only provides the platform for rendering terminal output and accepting keyboard input"

hterm/README.md

NaSSH Relay Protocol

- HTTP-to-SSH relay supported in Chrome Secure Shell
- Defines a series of HTTP handlers
- ack + SSH payload wrapped in a WebSocket frame
- Uses WebSocket binary frames (`0x82`), as opposed to UTF-8 frames (`0x81`)*


0
Ack offset
4
Payload
n

Acking Protocol

- Both the client and server keep track of the bytes read and written
- Upon connection the client sends the ack offset in the query string
- The server sends the contents of the retransmission buffer minus the ack offset

Retransmission Buffer

- The server keeps a buffer of the bytes received by the server backend
- This is trimmed when we receive the updated ack from the client

The Server

WebSockets + Nginx

```
server {
    # Allow WebSocket connections.
    location / {
        ...
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
    }
}
```

WebSockets + Cyclone

- We used Cyclone's built in WebSockets support, since our access proxy was already written using this framework
- Only a minor modification was required to get binary WebSocket messages working (4 loc)

/cookie
/proxy
/read
/write
/connect

5

HTTP handlers

/cookie

/proxy

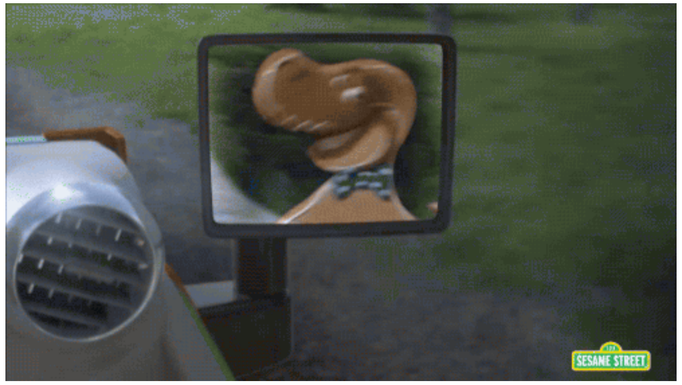
~~/read~~

~~/write~~

/connect

3

HTTP handlers



/cookie

Handles authN/authZ and does a 302 redirect to the Chrome extension ID or localhost

/proxy

Opens the TCP connection to the backend server and returns a `<UUID>` for identifying the connection

/connect

The actual WebSockets handler.
Handles bidirectional communication
between the client and server

The Client

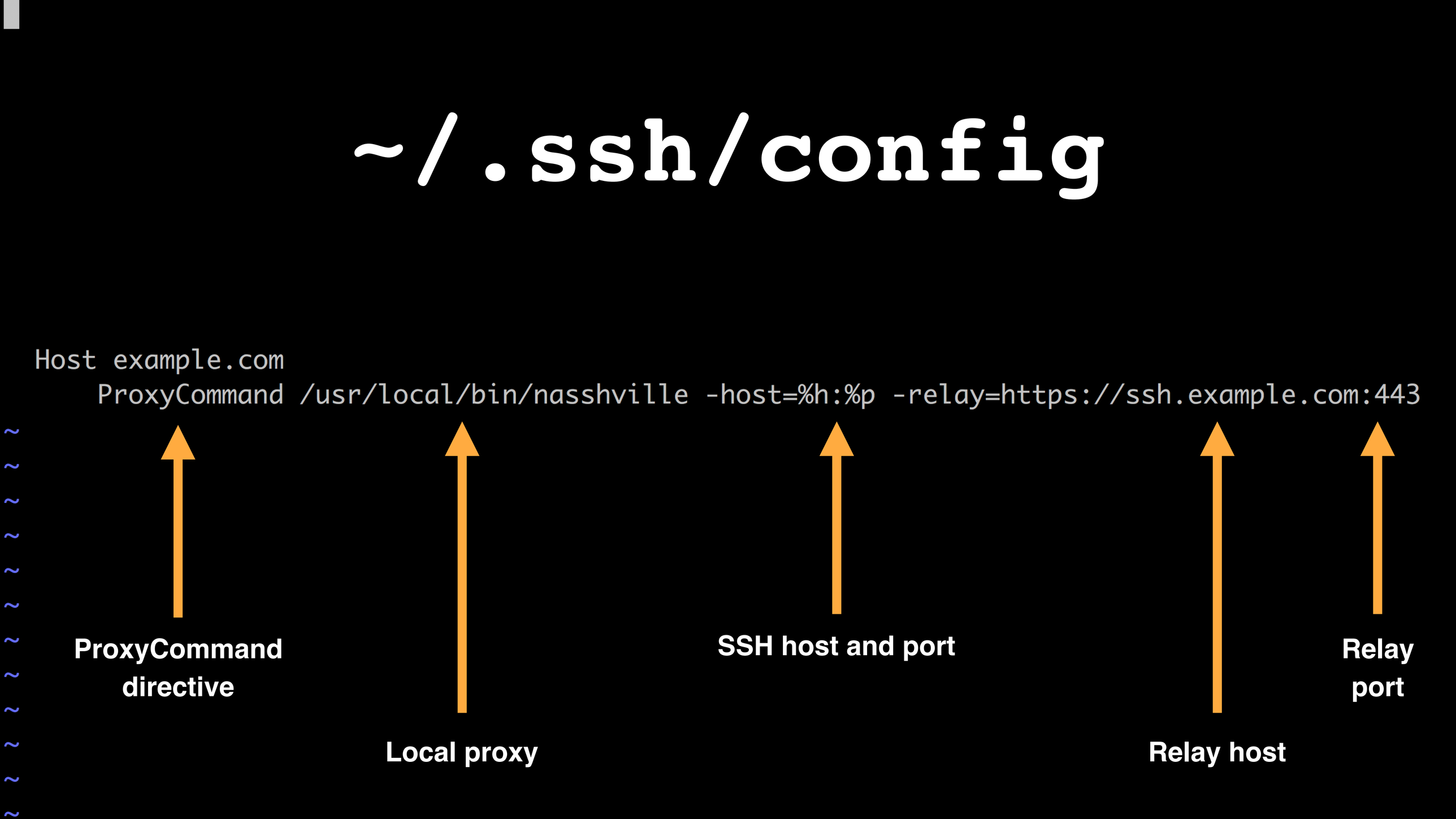
Chrome Secure Shell

Standard SSH Tooling (OpenSSH)

- Local, on-demand proxy written in Golang
- Point to the local proxy with the SSH `ProxyCommand` directive
- The local proxy is a WebSocket client
- `ProxyCommand` uses `stdin` and `stdout` to communicate with the local proxy and Terminal

~/.ssh/config

```
Host example.com
    ProxyCommand /usr/local/bin/nasshville -host=%h:%p -relay=https://ssh.example.com:443
```



Relay Host?

- Relay host is a DNS name that resolves to the access proxy IP (one per service)
- We needed to make a policy decision when the `/cookie` handler is hit, but the Chrome extension doesn't tell us what server we're connecting to
- In order to get per-server policies working this was necessary

Local HTTP Server

- In order to get the cookie used for authenticating through the access proxy, we start an ephemeral HTTP server for "catching" the cookie in the local proxy
- The server redirects to http://localhost:<port>/callback/?cookie=<cookie>&server=<server>&port=<port>&relay=<relay>
- `port` is an ephemeral port assigned by the OS that is passed to the access proxy in the query string

OpenSSH + NaSSH Relay, Step-by-Step

/cookie

- User types `ssh user@example.com`
- `ProxyCommand` launches the local proxy, passing it `stdout` from SSH as its `stdin`
- Local proxy hits https://ssh.example.com/cookie?path=callback&lport=<lport> to initiate SSO flow/2FA
- Server redirects to http://localhost:<lport>/<path>?cookie=<cookie>&relay=<relay>&server=<server>&port=<port>

/proxy

- Local proxy hits https://ssh.example.com/proxy?host=example.com&port=22
- Generate `sid` (`UUID`), tie to a `session` object which is kept in memory, establish TCP connection to service
- Keep track of the TCP connection in the `session` object, `dataReceived` callback fires when data is received from the backend service
- Return the `sid` in the response body

```
class TCPSession(object):

    def __init__(self,
                 protocol=None,
                 websocket=None,
                 ret_buf=b'',
                 read_count=0,
                 write_count=0):
        self._protocol = protocol     # Relay <-> server (TCP)
        self._websocket = websocket   # Relay <-> client (WebSockets)
        self._ret_buf = ret_buf
        self._read_count = read_count
        self._write_count = write_count
...
```

```
sessions = {}
...
session = TCPSession()
sid = str(uuid.uuid4())
sessions[sid] = session
...
```

/connect

- Local proxy hits wss://ssh.example.com/connect?sid=<uuid>&ack=0&pos=0
- Server responds with `HTTP 101 Switching Protocols`
- At this point callbacks fire on new connections and received messages
- The local proxy takes `stdin` from SSH and passes to the access proxy, whereas data received goes to `stdout`, (which is displayed in the Terminal thanks to `ProxyCommand`)

`connectionMade`

- Look up session ID provided in the query string
- Point to this WebSocket connection in the session object, (e.g., `self.session.websocket = self`)
- Update read_count and write_count on the session object

`connectionMade`, cont.

- Trim the retransmission buffer: `ret_buf = ret_buf[-ack:]`
- Send a WebSocket message to the client with the contents of the retransmission buffer

```
def connectionMade(self, *args, **kwargs):
    ...
    self.tcp_session = sessions.get(sid)
    self.tcp_session.websocket = self
    self.tcp_session.read_count = max(ack, self.tcp_session.read_count, 0)
    self.tcp_session.write_count = pos
    old_buf = self.tcp_session.ret_buf
    rc = self.tcp_session.read_count
    delta = ((rc & 0xffffff) - (ack & 0xffffff)) & 0xffffff
    self.tcp_session.ret_buf = b'' if delta == 0 else old_buf[-delta:]
    header_buffer = struct.pack('!i', self.tcp_session.write_count)
    payload = header_buffer + self.tcp_session.ret_buf
    self.sendMessage(payload, code=0x82)
```

`messageReceived`

- Update the write count to `len(message) - 4` (ignore 4 byte ack in message)
- Send `message[4:]` to the backend service
- Trim the retransmission buffer using the `ack` provided in the message by the client

```
def messageReceived(self, message):
    wc = len(message) - 4
    self.tcp_session.write_count += wc
    unseen_data = message[4:]
    self.tcp_session.protocol.transport.write(unseen_data)
    ack = struct.unpack('!i', message[:4])[0]
    old_buf = self.tcp_session.ret_buf
    rc = self.tcp_session.read_count
    delta = ((rc & 0xffffff) - (ack & 0xffffff)) & 0xffffff
    self.tcp_session.ret_buf = b'' if delta == 0 else old_buf[-delta:]
```

`dataReceived`

- Increment the read count: `read_count += len(data_received)`
- Concatenate `write_count` with `data`
- Send a WebSocket message to the client with the received data

```
def dataReceived(self, data):
    self.tcp_session.read_count += len(data)
    header_buffer = struct.pack('!i', self.tcp_session.write_count)
    payload = header_buffer + data
    self.tcp_session.websocket.sendMessage(payload, code=0x82)
    self.tcp_session.ret_buf += data
```

Misc.

Gotchas

- The Chrome Secure Shell extension automatically reconnects when it receives a close control frame, so to close the session permanently you send an empty payload with a negative ack
- Make sure the retransmission buffer is sent on `connectionMade`, otherwise race conditions can occur if the server responds before the client sends its version string

What About Other Protocols?

Other Protocols

- For non-SSH protocols that don't support `ProxyCommand`, a socket can be used instead, (e.g., RDP, VNC)
- The client connects to the local proxy and is responsible for tunneling bytes to the access proxy

References

BeyondCorp: A New Approach to Enterprise Security
BeyondCorp: Design to Deployment at Google
BeyondCorp: The Access Proxy
Migrating to BeyondCorp: Maintaining Productivity While Improving Security
ScaleFT BeyondCorp Summary
How Google Protects Its Corporate Security Perimeter without Firewalls
BSidesSF 2017 - BeyondCorp: Beyond “fortress” security
Google Cloud Identity-Aware Proxy
Open-Source Java NaSSH Relay
Open-Source NodeJS NaSSH Relay
Nginx auth_request

https://bit.ly/beyondcorp-ssh-proxy

Questions?

https://bit.ly/beyondcorp-ssh-proxy