Dynamic host configuration, please

Florian Obser
florian@openbsd.org

***Abstract*— Smartphones are always online devices in urban areas. They are even mostly online in rural areas. They deal with many different kinds of networks with only minimal configuration from the user. This paper will cover how we achieved a similar user experience on OpenBSD laptops. We will cover how we remember past visited Wi-Fi networks, automatically configuring IPv4 and IPv6 addresses and dealing with DNS in challenging network environments. We will also point out security measurements we put in place while dealing with untrusted networks.**

I. Join the Wi-Fi

When we bring a device to a never-before-visited network location we need the network name and password for the Wi-Fi or select an open Wi-Fi network.

Smartphones provide a user interface (UI) for this where we select the Wi-Fi from a drop-down list and then we are prompted for a password.

On OpenBSD, network interfaces are configured by ifconfig(8), or persistently in /etc/hostname.IF[1], which is read by netstart(8) during boot. netstart(8) calls ifconfig(8) internally to handle the network configuration. `# ifconfig iwm0 scan` will list the available Wi-Fi networks for the `iwm0` interface.

For a long time, we could only configure one Wi-Fi network:

```
$ cat /etc/hostname.iwm0
nwid home wpakey "trivial password"
inet autoconf
inet6 autoconf
up
```

This configures a Wi-Fi network named "home" and a password "trivial password". IPv4 and IPv6 auto-configuration are enabled. Whenever the network is in range the kernel automatically connects to it.

That is not a good user experience (UX). We typically take our laptops with us and connect to different Wi-Fi networks, like our smartphones. We have a Wi-Fi at home, at work, there are open Wi-Fis at hotels, and so on.

People came up with all kinds of unusual shell scripts that would run in the background or were triggered by cron(8) to notice when the laptop moved to a different Wi-Fi. The script would then call ifconfig(8) to reconfigure Wi-Fi from a list of networks it knew about. This was all fragile and unlike how OpenBSD works.

[1]IF denotes a specific network interface. For example for iwm0 the file is `/etc/hostname.iwm0`

Peter Hessler (phessler@), with the help of Stefan Sperling (stsp@) went ahead and tackled this problem: what if we could pass multiple `(name, password)` tuples to the kernel and the kernel would choose the right one?

```
$ cat /etc/hostname.iwm0
join home wpakey "trivial password"
join work wpakey zUDciIezevfySqam
join "Airport Wi-Fi"
join ""
inet autoconf
inet6 autoconf
up
```

`join` implements exactly this. The argument to `join` is the name of the network and the following `wpakey` is the password for that network. If we leave out the `wpakey`, the Wi-Fi is open and does not require a password. Using `join` with the empty string (`join ""`) means the kernel will try to connect to any open Wi-Fi if no Wi-Fi from the join-list is found first.

We still need to configure the name and password by editing a file in `/etc/` and run netstart(8) when we encounter a new Wi-Fi. This is probably not the best UI[2] but the UX is pretty good and on par with a smartphone. Once the Wi-Fi has been configured by adding a `join` line, the kernel will automatically reconnect to a known Wi-Fi whenever it comes within range.

II. Stop slacking

Now that we are connected to the Wi-Fi, we need to configure IP addresses.

We started our efforts to improve the network configuration user experience with IPv6 for two reasons. One is that IPv6 is still a technology for early adopters who are used to difficulties when using new technologies and are eager to help debug the problems that might arise.

Another reason for our work was the fact that OpenBSD got IPv6 support from the KAME project in the late 1990s and early 2000s after which it quieted down again. The network configuration was handled mostly in the kernel, so there was no isolation from malicious input. For the most part it assumed a stationary workstation that tried to acquire an IPv6 prefix for stateless address auto-configuration during boot by sending three router solicitations and then listened for router advertisements to create auto-configuration addresses and renewed their lifetimes when a new advertisement flew

[2]ed(1) is the pinnacle of UI design, as far as the author is concerned.

by. There was some rudimentary code in rtsold(8) to handle movement between networks, but it was rarely used since it was optional. rtsold(8) was used in one-shot mode where it would send at most three router solicitations when an interface connected to the network and then it would exit.

We started to write slaacd(8) and once that was working we could delete rtsold(8) and remove considerable pieces of code from the kernel.

slaacd(8) is a privilege-separated network daemon that build previous experience with privilege separation in OpenBSD. It uses three processes, (1) the *parent* process to configure the system, (2) the *frontend* process to talk to the outside world and (3) the *engine* process to handle untrusted data and run a state machine for the stateless address auto-configuration protocol.

pledge(2) restricts what a process is allowed to do and this is enforced by the kernel. Enforcement means that the kernel will terminate processes that violate what they have pledged to do. The pledges themselves are in broad strokes, we do not concern ourselves with single system calls but with groups of system calls. For example, the process is allowed to interact with open file descriptors (`"stdio"`), it is allowed to open connections to hosts on the Internet (`"inet"`), and it is allowed to open files for reading (`"rpath"`).

The *parent* process pledges that it will only open new network sockets, send those to other processes and reconfigure the routing table (`"stdio inet sendfd wroute"`). The *frontend* process pledges to only receive file descriptors, open unix domain sockets and check the state of the routing table (`"stdio unix recvfd route"`). Checking the routing table includes seeing which flags are configured per interface. The *engine* process pledges to only read and write to already open file-descriptors (`"stdio"`). The *engine* process is very restricted in what it is allowed to do. This is important because it handles untrusted data coming from the network. While the *frontend* process talks to the network, it never looks at the data. An attacker will not be able to confuse the *frontend* process with data they send. They can and have confused the *engine* process.

For more details see "Privilege drop, privilege separation, and restricted-service operating mode in OpenBSD".

slaacd(8) is enabled per default on all OpenBSD installations.

IPv6 stateless address auto-configuration is enabled on an interface by setting the `AUTCONF6` flag using ifconfig(8): `ifconfig iwm0 inet6 autoconf`. The kernel announces this changed interface flag to the whole system using a broadcast route message. slaacd(8) reads those messages using a route(4) socket.

slaacd(8) handles all aspects of stateless address auto-configuration. It sends router solicitations when needed, either multi-cast or uni-cast, depending on which is appropriate. It waits for router advertisements, parses them, and configures default routes, global and temporary IPv6 addresses, and passes name server information via a route message to the rest of the system. It takes care of the lifetimes of addresses, default routes, and name server information expiring and removes those from the system when no router advertisements are received to extend the lifetime.

slaacd(8) also monitors when network interfaces regain their connection to a network. For example because the laptop woke up from suspension or it got moved out of range of a Wi-Fi network and moved back within range. It then needs to find out whether it connected to the same network as before or whether it is now in a new network. If it is a new network we need to replace the old addresses, default route and name servers. If there is no IPv6 available it needs to remove the old information.

The stateless address auto-configuration specification allows multiple default routers being present on the same layer-two network, announcing the same or different network information. slaacd(8) tries to handle this, but this has not been extensively tested in all possible cases. There are still open questions being discussed at the IETF on how to run networks with different network prefixes in the same layer-two network.

slaacd(8) is able to handle multiple interfaces and we will show later how we pick the right source address when multiple addresses are available to choose from.

III. Dynamic host configuration, please

With IPv6 address configuration mostly solved, it was time to look at IPv4 again. We used a fork of ISC's dhclient(8). Henning Brauer (henning@) added privilege-separation to it and Kenneth Westerback (krw@) has been maintaining it in the past few years. However, the privilege-separation was never quite right which became more visible with the integration of pledge(2) and it turned out to be difficult to integrate some of the features we developed in slaacd(8).

It was time to write a new daemon and Otto Moerbeek (otto@) came up with a name for it: dhcpleased(8). It is pronounced as "dynamic host configuration, please" with the "d" silent.

On a very high level, IPv4 DHCP and IPv6 stateless address auto-configuration are very similar. We request some information from the router[3], we use it to configure the system and we make sure that information does not expire. When we move networks we need to probe whether our information is still up to date and if not, reconfigure the system.

We opted for the obvious solution, which is to copy `sbin/slaacd` to `sbin/dhcpleased` and replace the IPv6 specific bits with IPv4 specific bits.

On paper DHCP looks more complicated than IPv6 stateless address auto-configuration because it negotiates with the server and there is a complicated state machine to implement.

In practice it is the other way around. The "stateless" part in IPv6 does not apply to the client. The client must keep state and implement a state machine to keep track of which routers are available and when various information expires. In IPv4 we talk to one server and all information expires at the same time.

[3]In IPv6 we might not need to request the information, it might just show up unannounced.

We will talk about a few differences between slaacd(8) and dhcpleased(8) in a moment, but from the user perspective both behave in the same way. They make sure that the address configuration and default gateway are always up to date and they pay attention when the machine moves between networks, either while awake or while sleeping.

Because dhcpleased(8) has to use bpf(4) instead of regular sockets for some of the network packets it needs to send, the *parent* process cannot use pledge(2). Currently, there is nothing it could pledge that would allow the usage of bpf(4). To protect the system and prevent exfiltration of sensitive data we use unveil(2) to restrict the *parent* process' view of the file system. dhcpleased(8) can only read its configuration file, read and write `/dev/bpf`, and read, write and create files underneath `/var/db/dhcpleased/` to store information about received leases.

While we could get away with not implementing a config file for slaacd(8), this did not work for dhcpleased(8). Some systems out there will only give us a DHCP lease if we send the correct *client id*, for example.

There are many DHCP options specified in RFC 2132. We have only implemented the bare minimum, only the options we need and can handle. We do not need a swap server or a cookie server, to name a few.

Like slaacd(8), dhcpleased(8) is enabled on all OpenBSD installations.

IV. Route priorities

dhcpleased(8) and slaacd(8) can handle multiple interfaces at the same time. The routing table might look like this:

```
$ netstat -nrf inet \
  | awk "{print $1,$2,$7,$8}"
Routing tables

Internet:
Destination  Gateway         Prio Iface
default      192.168.1.1        8 em0
default      192.168.178.1     12 iwm0
[...]
```

We end up with two default routes, one gateway is reachable via the *em0* interface with priority value 8 and the other gateway is reachable via the *iwm0* interface with priority value 12. A route has higher priority when its priority value is lower. *em0* is an Ethernet interface and it gets higher priority over the Wi-Fi interface *iwm0*. All things being equal, the kernel will pick the address from *em0* as source address when making a new connection to the internet and route traffic over the Ethernet interface, which is presumably faster.

If we pick up the laptop and unplug the Ethernet interface, the route over *em0* is no longer usable and existing connections using it will stall and time out. New connections will instead use *iwm0*.

If we plug the Ethernet interface *em0* back in, the session might come alive again and new connections will use *em0*. Connections that are running over *iwm0* will continue working, because the interface is still connected to the Wi-Fi.

Applications like web browsers, email clients or even video conferencing systems will automatically establish a new connection when they notice that the old one is dead.

Unfortunately ssh(1) is not one of them. If switching between wired and wireless happens rarely, tmux(1) on the remote system might help with ssh(1) disconnects, or a wg(4) tunnel can be used so that the source address does not change when switching between wired and wireless.

V. Cellular networks

In addition to Ethernet and Wi-Fi networks, OpenBSD supports "Mobile Broadband Interface Model" devices using the umb(4) driver. These can be used to connect to UMTS or LTE networks. They require a SIM card and after being configured using a PIN they will connect to cellular networks and automatically configure an IP address and default route. The default route has an even lower route priority than Wi-Fi so it will only be used when Ethernet and Wi-Fi are not connected.

VI. It is always DNS

Humans are not particularly good at remembering addresses like `2606:2800:220:1:248:1893:25c8:1946` and are much better with names like *example.com*. When we run `ping6 example.com` we will end up in libc's stub resolver. It will open `/etc/resolv.conf` and look for *nameserver* lines to use for DNS resolution.

We can learn name servers from dhcpleased(8), slaacd(8), umb(4), and iked(8). Historically dhclient(8) owned `/etc/resolv.conf`, which means that no other process could add name servers to it. dhclient(8) would just overwrite whatever was in there whenever it renewed its lease. This made it impossible to sometimes move to an IPv6-only network. slaacd(8) could not configure name servers and the left-over IPv4 name servers were not reachable.

We can either teach all name server sources to somehow cooperate and share responsibility of `/etc/resolv.conf` or we can run an arbitrator that collects name servers from diverse sources and handles the contents of `/etc/resolv.conf`.

resolvd(8) is such an arbitrator. It is another always enabled daemon. It collects name servers from all the mentioned sources and adds them to `/etc/resolv.conf`.

It also monitors if `/etc/resolv.conf` gets edited in which case it rereads the file and makes sure that the learned name servers are at the beginning of the file. This is useful when the administrator of the machine decides to add options to `/etc/resolv.conf`. For example, we can edit the file and add `family inet6 inet` to prefer IPv6 over IPv4 and resolvd(8) will cope. There is no need for an extra configuration file, `/etc/resolv.conf` is the configuration file.

Name servers are announced using route messages and resolvd(8) listens for them using a route(4) socket. They can also be observed using the route(8) tool: `$ route monitor`.

resolvd(8) can also request that name servers are re-announced by their sources. This is useful when resolvd(8) gets restarted.

VII. Let us unwind a bit

Plain DNS is not a secure protocol. It exchanges unauthenticated UDP packets without any integrity protection. This makes it easy for an attacker to spoof answer packets.

DNS answer packets are untrusted data, they come from the network. However, the process that sends DNS queries and parses the answer using the libc functions is almost always the single main process of the tool. When we run `ping example.com`, DNS packets are parsed using our user. An attacker who can spoof a DNS answer might be able to trigger a bug in libc and gain code execution that way.

On OpenBSD, ping(8) pledges `"stdio DNS"` so the attacker will not get very far, but there are many more programs in ports that are not pledged that might want to resolve names.

It would be worthwhile to have some sort of proxy running on localhost such that DNS packets from the outside need to traverse a well locked down process running in a different address-space and as a different user than the program that needs to resolve a name.

An early experiment was rebound(8), written by Ted Unangst (tedu@). It was simplistic and did not understand DNS at all, it would just forward packets, but it would sit between the Internet and the program.

An alternative is to run a full recursive resolver like unbound(8) on the laptop, but this leads to problems, too. unbound(8) expects a well working network where nothing interferes with DNS, this is true in data centres and can be achieved in well maintained home networks, but it is not something we find when moving laptops to arbitrary networks like free Wi-Fi in a hotel or at an airport.

It turns out that often the quality of the network changes over time. When we first connect to a hotel Wi-Fi we may find ourselves in what is referred to as a *captive portal*. Everything is blocked, DNS gets intercepted, and we are redirected to a website where we need to agree to the terms and conditions and maybe provide our name and room number. Once we are past that, network quality improves considerably and we are mostly free to talk to the outside world.

This is where unwind(8) comes in. It is another privilege-separated network daemon that provides a recursive name server for the local machine. resolvd(8) detects when it is running and automatically rewrites `/etc/resolv.conf` to have only `nameserver 127.0.0.1` listed as name server.

This solves or improves upon the first problem. Programs that need DNS resolution are insulated from the Internet. An attacker needs to get past unwind(8) first before they can try to attack the libc stub resolver.

unwind(8) understands and speaks DNS and it actively observes the network quality.

We did not write our own recursive name server. That would be difficult and since DNS is constantly evolving, it would also require extra work to keep up. Instead we decided to use libunbound, which is part of unbound(8). It is developed under a BSD license by NLnet Labs.

The resolver process pledges `"stdio inet dns rpath"` and restricts access to the file system using unveil(2) to `/etc/ssl/cert.pem`. This is the process that is exposed to the Internet and handles untrusted data. It would be preferable to have one process exposed to the Internet and another to parse untrusted data but that is not possible to do with libunbound.

Since we are using a real recursive name server, that gives us many options on how we can resolve names:

- We can do our own recursion, walk down from the root zone using qname minimization to improve privacy.
- We can use the name server we learned from dhcpleased(8) and slaacd(8) as forwarders, so we do not need to do our own recursion, which might be faster.
- We can try to opportunistically speak DNS over TLS (DoT) to the learned name servers to prevent eavesdroppers from listening in.
- We can configure forwarders manually to not depend on the network provided name servers. Those might be more trustworthy. They can also be DoT forwarders to prevent eavesdropping.
- As a last resort, unwind(8) can behave exactly like the libc stub resolver[4].

We call these resolving strategies and unwind(8) actively probes if they are usable by sending test queries when it notices that the network changed, for example because the laptop moved to a different Wi-Fi network or woke up from suspension. It then orders them by quality and picks the best one.

There is an implicit skew in the strategies for finding the best one: a manually configured DoT name server is always considered better than a name server provided by the local network, as long as it is available and not too slow.

unwind(8) is not too concerned about preserving privacy, it is pragmatic and tries to resolve names the best way it can, and it will use the local name servers provided by the network if those are the only ones available.

Since unwind(8) uses libunbound it also supports DNSSEC. DNSSEC provides data integrity and cryptographic authenticity, it does not provide confidentiality.

unwind(8) is pragmatic about DNSSEC. When it tests the quality of a resolving strategy it also tries to find out if DNSSEC is available. There are many reasons why DNSSEC might not be available: the network is misconfigured, DNSSEC is blocked or the laptop does not (yet) have the correct time. If DNSSEC does not work, unwind(8) does not insist on using it.

Of course this makes it susceptible to a downgrade attack. To mitigate this, unwind(8) will insist on DNSSEC working after it has discovered once that DNSSEC is working in the

[4]Call this the "Dutch train problem": the free Wi-Fi on Dutch trains do not like DNS queries with an *EDNS0* option, they intercept them, do not understand them, and answer *NXDOMAIN*. There are other free Wi-Fi networks that are similarly broken.

local network. This means that an attacker needs to be able to block DNSSEC from the moment we connect to a network. They cannot show up later and try to downgrade us. unwind(8) will only become lenient again when we connect to a new network.

This is not a strong mitigation of course, but DNSSEC is not a fix for everything at the resolver. Applications also need to do their part and decide how much they are willing to trust DNS. For example ssh(1)'s *VerifyHostKeyDNS* feature will only trust host key fingerprints it obtained from DNS if they were validated using DNSSEC and the validator runs on the local machine[5]. Otherwise it will ask the user what to do.

A worst-case scenario when joining a partially broken Wi-Fi network with captive portal and a manually configured DoT name server might look like this:

1) We connect to the network, we cannot reach the DoT name server and cannot do our own recursion.
2) unwind(8) will choose the name server provided by the network. It also notes that we just connected to a new network so it is lenient with respect to DNSSEC validation. In effect it will ignore validation errors.
3) We try to access a website and the captive portal detection in the browser triggers. We click the buttons and fill in the forms until we are allowed on the internet.
4) unwind(8) notices that it can do its own recursion.
5) At the same time, unwind(8) notices that the DoT name server is also reachable now and starts using it.

unwind(8) does not natively support DNS over HTTPS (DoH) and we sometimes find ourselves in networks that block everything except for TCP port 443. One way around this is to use dnscrypt-proxy from ports which does support DoH. We can point unwind(8) at it by manually configuring a plain DNS forwarder in addition to a DoT forwarder:

```
$ cat /etc/unwind.conf
forwarder "9.9.9.9" port 853 \
  authentication name "dns.quad9.net" DoT
forwarder "2620:fe::9" port 853 \
  authentication name "dns.quad9.net" DoT
# dnscrypt-proxy for DoH
forwarder "127.0.0.1" port 5353
```

VIII. Time for gelato

There are various transition technologies that get us from an IPv4-only Internet to an IPv6-only Internet. We will only look at *NAT64*, *DNS64*, and *464XLAT*.

NAT64 allows us to reach IPv4 hosts from an IPv6-only network by pretending that the hosts are IPv6 enabled. IPv6 addresses are so large that we can easily encode all of IPv4 in an IPv6 /64 prefix, which is the usual size of on IPv6 prefix we see per layer-two network. In fact we don't need the whole /64, a /96 is enough to encode the whole IPv4 Internet.

Let us pretend we know the /96 prefix used for *NAT64* and the IPv4 address we want to reach. Forming an IPv6 address for the host is then simply a bitwise-or operation of the IPv4 address with the /96 prefix, the IPv4 address fills in the lower bits of the IPv6 prefix. This is called address synthesis.

We can then use this address to connect to the IPv4-only host. Somewhere on the network path is the *NAT64* gateway that is dual stacked. It knows that our packets are using *NAT64* because it is configured with the /96 prefix. It intercepts the packets and forms IPv4 packets and sends them on their way. The gateway needs to be stateful to be able to *NAT* the return traffic back to us.

We use DNS to find out the IPv4 address that we want to connect to. The local name servers that slaacd(8) learned about would know about the *NAT64* prefix used in the network and do the address synthesis for us. This is called *DNS64*. The problem with this is that the name servers spoof DNS answers, something that DNSSEC tries to prevent. unwind(8) will detect this and generate an error, or unwind(8) might not even talk to the designated name servers at all.

To get around this, unwind(8) itself can detect the presence of *DNS64* on a network by asking the local name servers for the *AAAA* record, i.e. the IPv6 address, for something that is guaranteed to never have one: *ipv4only.arpa*. If it gets an answer, it can reverse the address synthesis and learn the *NAT64* prefix. With that information it can do *DNS64* itself and there is no longer a problem with DNSSEC.

The downsides of this mechanism are that it is quite complicated, it messes around with DNS, and it does not work with IPv4 address literals. It also does not work with programs that are fundamentally IPv4-only: `ping example.com` will never work in an IPv6-only network with only *NAT64 / DNS64*.

Instead of pretending that the IPv4 host we want to reach has IPv6, we can pretend to have working IPv4 if a *NAT64* gateway is present. We ask the kernel via the pf(4) firewall to do the IPv4-to-IPv6 translation for us. The *NAT64* gateway will then do the reverse translation and send an IPv4 packet on its way. This is called *464XLAT*.

We first need an IPv4 address, RFC 7335 reserved `192.0.0.0/29` for this purpose:

```
ifconfig pair1 inet 192.0.0.4/29
```

We then need a default gateway:

```
ifconfig pair2 rdomain 1
ifconfig pair2 inet 192.0.0.1/29
```

Because pf(4) will only do address family translation on inbound rules we need a different *rdomain* and use pair(4) interfaces. We need to connect them:

```
ifconfig pair1 patch pair2
```

And then we can configure our default route:

```
route add -host -inet default 192.0.0.1 \
  -priority 48
```

We set it to a very low priority[6] so that it does not interfere

[5]Technically not entirely true, ssh(1) trusts what libc indicates and libc automatically trusts localhost. See *trust-ad* in resolv.conf(5).

[6]Remember that a high priority **value** means low priority.

with routes that dhcpleased(8) configures when we move to an IPv4 enabled network.

We then need to configure address family translation in pf(4) when we detect *NAT64* being present. This is were gelatod(8) comes in. It is a Customer-side transLATor (*CLAT*) configuration daemon. *CLAT* is what *464XLAT* calls the address translation happening on the laptop.

gelatod(8) is yet another privilege-separated daemon[7] that checks for the presence of a *NAT64* gateway whenever we change networks. It does so either via the *ipv4only.arpa* trick or explicitly via router advertisements. RFC 8781 specifies how a network can signal the presence of a *NAT64* gateway.

gelatod(8) needs a pf(4) anchor into which it adds rules that are similar to this example:

```
pass in log quick on pair2 inet \
  af-to inet6 \
  from 2001:db8::da68:f613:4573:4ed0 \
  to 64:ff9b::/96 \
  rtable 0
```

The rule is doing address family translation to IPv6 on incoming packets on `pair2`. In this example it uses `2001:db8::da68:f613:4573:4ed0` as the IPv6 source address, gelatod(8) learned this from the system when slaacd(8) configured it. `64:ff9b::/96` is the learned *NAT64* prefix and we are moving traffic back to `rtable 0`. Remember `pair2` is in rdomain 1.

While this works rather well, it is also complicated to set up, which is why gelatod(8) is not in OpenBSD base but lives in ports. We believe in good defaults in OpenBSD and try to make it easy for the user.

IX. Future work

We would like to have the functionality of gelatod(8) in OpenBSD base. gelatod(8) was mostly a proof of concept and we imagine that a new network device like clat(4) would take over the role of client side address family translation. It could be always present and gelatod(8) would just enable and disable it. At that point we could move the functionality into slaacd(8) and delete gelatod(8). *CLAT* is defined as a stateless mechanism so it does not need the full pf(4) machinery for address family translation.

It would be valuable to have DNS over HTTPS (DoH) and DNS over Quic (DoQ) natively in unwind(8). We are mostly waiting on upstream to implement support in unbound(8).

There are also minor issues that could be improved:

- The captive portal detection in unwind(8) is not perfect and could be improved upon.
- dhcpleased(8) and slaacd(8) should remember IP addresses from networks they have been connected to previously, to be able to quickly re-establish connectivity by probing whether we are connecting to a previous network while the lifetime of our addresses have not expired yet. RFC 4436 "Detecting Network Attachment in IPv4 (DNAv4)" and RFC 6059 "Simple Procedures for Detecting Network Attachment in IPv6" discuss the details.
- It would be helpful if the dhcpleased(8) parent process could be pledged. This is currently not possible because of bpf(4). Things to investigate here are changes to the network stack that would allow us to use raw sockets instead of bpf(4) sockets or the ability to dup(2) an existing bpf(4) socket and reprogram the interface it is using.

[7] At this point we will not go into pledge details.

X. Conclusion

In this paper we have described how OpenBSD improved the user experience and security of laptop users when visiting diverse network locations. The system remembers Wi-Fi networks and automatically connects to them. It automatically discovers when the network changes and acquires new IPv4 and IPv6 addresses or renews existing configurations. OpenBSD also actively probes available DNS resolving strategies and picks the best one available. Privilege-separation and restricted service operating mode ensure that untrusted data is parsed with the least privileges necessary, protecting the rest of the system.

Acknowledgment

The author would like to thank Mine Temuerhan for copy-editing the paper.