USENIX Association

Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

© 2002 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Authorization and charging in public WLANs using FreeBSD and 802.1x

Pekka Nikander

Ericsson Research NomadicLab
pekka.nikander@ericsson.com

Abstract

The IEEE 802.1x standard defines a link-layer level authentication protocol for local area networks. While originally designed to authenticate users in a switched Ethernet environment, it looks like the most important need for 802.1x lies in wireless networks, especially IEEE 802.11b based Wireless LANs. Furthermore, due to the flexibility of the Extensible Authentication Protocol (EAP), the heart of 802.1x, it looks like 802.1x could be used for many purposes its original designers have not foreseen.

In this paper, we describe an FreeBSD-based open source 802.1x implementation, and show how it can be used to implement different authorization and charging systems for public WLANs, including a pre-paid, pay-per-use charging system and another one based on community membership. The implementation is based on the netgraph facility, resulting in a surprisingly flexible and simple implementation.

1 Introduction

With the advent of 802.11 based Wireless Local Area Networks (WLANs) and the proliferation of laptops and PDAs, the usage of Ethernet networks is changing. What once was meant to statically connect computers belonging to a single organization, is now increasingly being used to provide Internet connectivity for mobile professionals. This change is very visible at certain high profile telecommunication and computer science conferences, including IETF meetings, where an increasing number of people are utilizing the wireless networks provided by the organizers.

The changing usage patterns brings forth new security problems. From the network point of view, the most prominent problem is that of access control. The network must somehow decide if and how to allow a network node to utilize the resources provided by the network. The IEEE 802.1x standard [1] is designed to provide a solution framework for this problem.

The 802.1x standard defines a method to run the Extensible Authentication Protocol (EAP) [2] in raw (non-IP) Ethernet frames. The resulting protocol is called EAP over LAN (EAPOL). The EAP protocol was originally designed as a flexible authentication solution for modem connections using the Point-to-Point Protocol (PPP). Now 802.1x is extending the same architecture to LANs, both wireline and wireless.

At the basic level, the 802.1x architecture, properly augmented with the RADIUS [3], allows the existing PPP based authentication, authorization and accounting infrastructure to be used to control LAN access in addition to PPP access. The solution works well in current environments, where the network providers and network users have an existing, pre-established relationship. However, if we consider public WLAN offerings at airports, hotels, cafes, and even offices and homes, it becomes more common than before that the prospective network user has no relationship with the provider. Thus, something new must be introduced, something that allows such a relationship to be build on the fly.

In this paper, we show how it is possible to set up a public WLAN environment that supports various kinds of authorization and charging schemes, including community membership based and pre-paid, pay-per-use accounts. The new schemes allow more flexible user–provider relationship management than the existing ones. The implementation is based on IEEE 802.1x, and the prototype runs under FreeBSD. The architecture is geared to small service providers, eventually scaling down to the level of individuals offering WLAN based Internet access through their xDSL or cable modem.

The rest of this paper is organized as follows. In Section 2, we briefly describe the IEEE 802.1x standard and the FreeBSD netgraph facility. After that, in Section 3, we describe our 802.1x implementation in broad terms, and in Section 4 we cover the implementation details. In Section 5 we describe how the implementation can be configured to support different usage scenarios, including pre-paid pay-per-usage accounts and community membership based accounts, and Section 6 includes our initial performance measurements. In Section 7, we discuss a number of open issues and future possibilities related to this ongoing work. Finally, Section 8 includes our conclusions that we have been able to achieve so far.

2 Background

Our work is mostly based on existing technologies, integrating them in a novel way. The only significant piece of new software that we have written is the base 802.1x EAPOL protocol [1], which is implemented as a pair of FreeBSD netgraph [4] nodes. Since we cannot assume that all readers are familiar with the IEEE 802.1x standard and the netgraph facility, we introduce them briefly in this section. Additionally, we discuss related work.

2.1 IEEE 802.1x

IEEE 802.1x [1] is a forthcoming standard for authenticating and authorizing users in Ethernet like local area network (LAN) environments. It is primarily meant to secure switched Ethernet wireline networks and IEEE 802.11 based WLANs. In the typical usage scenarios, the network requires user authentication before any other traffic is allowed, i.e., even before the client computer is assigned an IP address. This allows corporations to strictly control access to their networks.

The 802.1x overall architecture is depicted in Figure 1. The central point of the architecture is an Ethernet switch or WLAN access point. Now, instead of directly connecting clients to the network, the access points contains additional controls, basically preventing the clients from communicating before they have positively authenticated themselves. The authentication takes place between an EAPOL supplicant, running on the client, and a background authenticator server. This is denoted with the thick, dashed arrow line. The authentication protocol can be any supported by the Extensible Authentication Protocol (EAP) standard [2].

Even though the actual authentication protocol is run between the supplicant client and the authentication

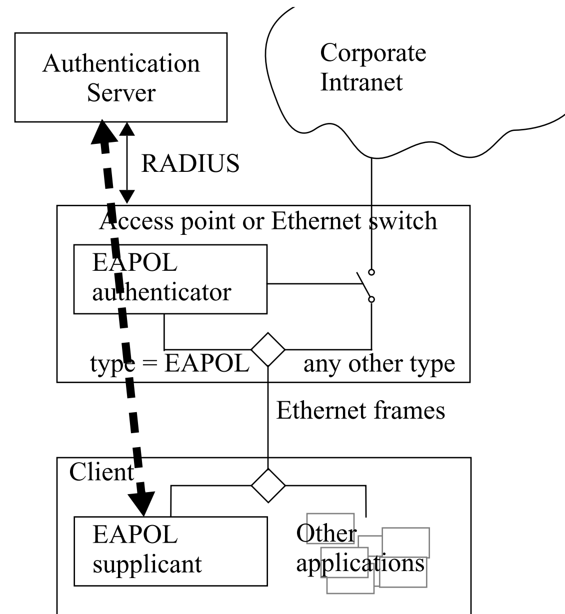

Figure 1: Basic IEEE 802.1x architecture

server, the protocol is mediated by an EAPOL authenticator function located at the access point. The EAPOL authenticator takes care of two functions. First, it mediates the EAP packets between RADIUS, used towards the authentication server, and EAPOL, used towards the client. Second, it contains a state machine that “snoops” the EAP protocol, learning whether the authentication server was able to authenticate the user identity or not. If the user was authenticated, it permits the client to freely communicate; otherwise the client is denied access from the network. In the figure, this latter function is depicted with the on/off switch within the access point.

It is important to note that 802.1x implements only authentication and MAC address based access control. Since MAC spoofing is fairly easy, the resulting system, as such, might not be secure enough. The need for additional security measures depends on the usage scenario; see Section 7.1 for the details.

2.2 EAPOL

The EAPOL protocol is run between the EAPOL supplicant, running on the client, and the EAPOL authenticator, running on the access point. It is a fairly simple protocol, consisting of a packet structure and state machines running on both the supplicant and the authenticator. The packets are based on raw Ethernet frames with little additional structure, as depicted in Figure 2, on the next page.

The protocol is typically initiated by the authenticator as soon as it detects that a new client has been connected

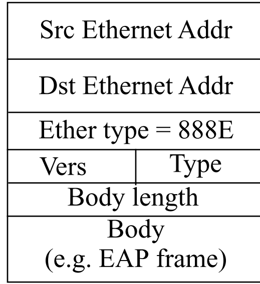

Figure 2: EAPOL packet structure

to an switched Ethernet port or that a new client has joined to a WLAN access point. However, the protocol can also be triggered by the client by sending an EAPOL start packet.

A typical protocol run is depicted in Figure 3, below. The first message is an EAP identity request, sent by the EAPOL authenticator. The supplicant replies with an EAP identity response, which is passed to the authenticator server. The authenticator server replies by running one of the possible EAP subprotocols within EAP request and response frames. Once the authentication phase has been completed, the authentication server sends an EAP success (or failure), indicating that the user identity was verified (or could not be verified). The EAPOL authenticator notices this message, and allows (or disallows) the client to communicate.

In addition to the base authentication, EAPOL also allows periodic re-authentication and logout by the client. These are most useful in a setting where connection time is used as a basis for accounting.

2.3 Netgraph

Netgraph [4] is a flexible network protocol architecture implemented in the FreeBSD kernel. Currently, it allows different link-layer protocols to be stacked between the device drivers and the network layer protocols, i.e., IP. Its basic benefit is flexibility; it makes it easy to add new link-layer subprotocols, and to stack different protocols,

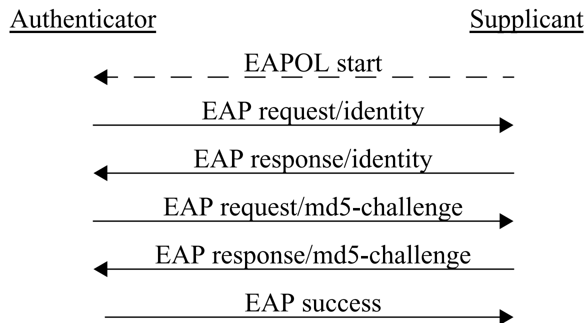

Figure 3: A typical EAPOL protocol run

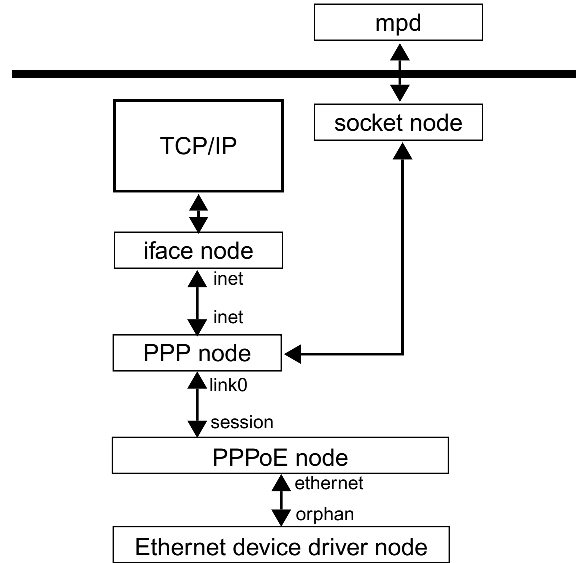

Figure 4: A netgraph stack implementing PPPoE

such as HDLC, Frame Relay, and PPP, in different ways. Furthermore, it allows intelligent filtering and pseudo-interfaces, thereby making it possible to make a difference between LAN clients based on their MAC addresses.

From the implementor's and administrator's point of view, netgraph is extremely flexible. Firstly, new protocol nodes are implemented as loadable kernel modules, making development and testing fairly easy. Secondly, the stacking order and connections between the protocol nodes are configured with a user level program. This makes it possible to use the same protocol nodes in many different ways. (See the examples in Section 5.)

A simple netgraph stack is shown in Figure 4. The stack is used to run PPP over Ethernet (PPPoE), a networking standard that a number of xDSL and cable operators use today. The actual PPP signalling is performed at the user level, using a separate daemon (mpd). However, the data packets flowing between the IP protocol and the physical interface are completely handled at the kernel level.

2.4 Related work

Even though there are a number of commercial 802.1x implementations available, the only other open source implementation that the author knows of is one by the Open1x project at University of Maryland, College Park [5]. Compared with our project, there are two major differences. Firstly, Open1x is a Linux based user level implementation while ours relies on netgraph. Secondly, based on the first source code release, the Open1x project seems to be more research oriented, trying to

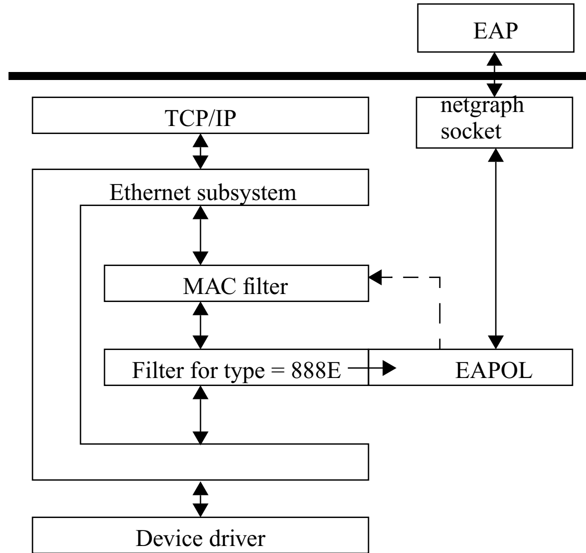

Figure 5: A netgraph stack implementing 802.1x compliant access point

identify security problems in the specifications, while our goal has been to produce a high quality implementation that could be used in production environments.

3 Software architecture

In this section, we describe our solution architecture in broad terms. The details are left for the next section, and a couple of usage scenarios are given in Section 5. In this section, we first describe the overall structure of the software, and then briefly describe the components.

3.1 Overall structure

Our 802.1x implementation consist of two new netgraph nodes, plus a number of user level programs. The user level programs implement the individual EAP subprotocols. The larger of the netgraph nodes implements the EAPOL protocol, providing a clean interface to the user level EAP programs, while the smaller netgraph node implements a simple MAC address filter.

A basic netgraph structure, used to implement a simple 802.1x compliant access point, is depicted in Figure 5, above. In this configuration, the 802.1x implementation is hooked inside the Ethernet subsystem (`if_ethersubr.c`) using suitable negraph hooks.

If we consider incoming packets, they are first routed to the first part of the EAPOL module. If the ethertype is `0x888E` (EAPOL) the packets are passed to the second part of the EAPOL node. Otherwise they are passed upward to the MAC filter. The MAC filter either passes or drops the packet, depending on the sender's MAC address. The list of passed addresses is maintained by the EAPOL module by sending events to the MAC filter. The passed packets are simply routed back to the Ethernet subsystem, which then passes the packets to the upper layer protocol.

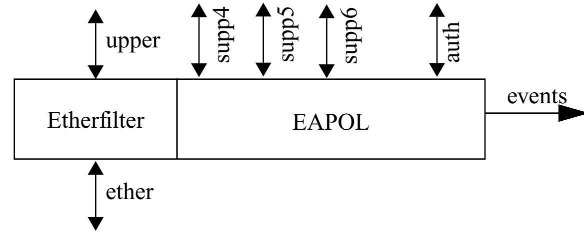

Figure 6: EAPOL node hooks

3.2 EAPOL netgraph node

The EAPOL netgraph node implements the EAPOL protocol at the kernel level. It includes both EAPOL authenticator and supplicant functionality, and is able to run both functions in parallel. However, it handles directly only the EAP identity and notification request/response exchanges. All actual authentication protocols must be implemented outside the EAPOL node. The node just includes hooks that different EAP protocol implementations can attach to. Due to the flexibility of netgraph, the protocols can either run inside the kernel, as netgraph nodes, or at the user level.

The external interfaces of the EAPOL node are described in Figure 6, above. The node filters received Ethernet frames and passes non-EAPOL frames to `upper` hook. Any packets received at the `upper` hook are passed to the `ether` hook. The `events` hook reports EAPOL related events, such as EAPOL start and logoff frames and EAP success and failure packets.

The `supp4`..`supp`*N* hooks are used to connect supplicant EAP modules to the EAPOL node. This architecture allows different EAP subprotocols to be implemented by different programs. For example, if one wants to use EAP TLS [6] for user authentication and at the same time EAP OTP [2][7] for client machine level access control, it is possible. In that case, there would be an EAP TLS program, including a GUI, which connects to `supp7`, and an EAP OTP daemon program, connecting to `supp5`. The TLS program would receive all EAP packets containing TLS related requests, and the OTP program those containing OTP related requests.

We originally planned to have a similar structure at the authenticator side, too. However, it turned out to be hard to keep track of sent requests and received responses so that the response packets could be delivered to the right EAP authenticator. The reason for this is that the tracking is based on the 8 bit EAP message ID code in the requests and matching responses. To support several parallel authenticators, it would be necessary to select unique message IDs to the outgoing requests, and to replace the message IDs with the original ones as responses are received. Since this functionality is usually not needed, the added complexity and kernel level state just seemed unnecessary.

Consequently, the author revised the base EAPOL module so that it supports only one authenticator hook and delivers all received EAP response packets to this hook. This serves very well integrated authentication systems where there is a back-end server. It is still possible to reintroduce the original structure by implementing an additional netgraph node that keeps track of outstanding requests and performs the necessary ID conversions.

3.3 User level EAP programs

As already discussed, the architecture allows the different EAP authentication protocols to be implemented as distinct user level programs. At the client (supplicant) side, the program simply receives EAP requests from the EAPOL netgraph node, processes them, and sends back EAP responses. If desired, the program can easily implement a user interface to request information from the user.

At the server (authenticator) side the situation is slightly more complex. Firstly, the authenticator must keep track of outstanding requests, based on the message IDs. Secondly, a single program usually wants to support several different authentication protocols, for example, since it wants to connect to a back-end authentication server. Furthermore, the program may need additional information about the client, such as the MAC address and the physical interface through which the packet was received.

The first two needs are covered by the prototype system. However, we do not currently pass information about the interface, nor the MAC address, to the user level. In the simple scenarios it is not needed, since the required information is passed directly within the kernel between the modules. On the other hand, we are planning to modify `ng_socket` so that it is able to pass metadata to the user level and vice versa. Once that facility is available, it becomes possible to pass the MAC address and interface index in `sockaddr_dl`. That would even allow future interoperability with PPP EAP.

In the simplest case, a server side EAP program simply receives EAP responses from the EAPOL netgraph node, converts them into RADIUS requests, and sends them to the RADIUS server. Similarly, it receives EAP requests within RADIUS packets from the RADIUS server, converts them back, and sends down to the EAPOL module. The EAP program does not need to take care of retransmissions, since the underlying EAPOL state machine takes care of that.

An alternative design would be to use locally authentication data, such as passwords or OTP keys. In that case, the authenticator module would receive an EAP identity response, look up the user from the local authentication database, create state, send an appropriate EAP request, and wait for a response. Once it receives a matching response, it would compare the received authenticator against the expected one, and either pass or reject the user. For an example of such a module, see Section 4.3.

3.4 MAC filter netgraph node

In addition to supporting the EAPOL protocol, an 802.1x authenticator also needs to include the functionality that blocks unauthenticated clients from communicating with the network behind the access point. To support this, we have implemented another netgraph node, the MAC filter. The MAC filter allows incoming packets to be routed to different hooks depending on the source MAC address. If there is no matching hook, the packets are passed to a default hook. If no other node is connected to the default hook, the packets are simply dropped.

In addition to allowing packets to be screened by their source MAC address, as required by base 802.1x, the design also supports more sophisticated modes of operation. Basically, different clients can be classified in different categories, and handled appropriately. This allows different service classes for different packets and direct bridging some packets to an Ethernet-level tunnel while other packets are allowed to reach the local IP routing layer. One possible scenario, multioperator support, is described in briefly in Section 5.3.

4 FreeBSD implementation

In this section we describe the implementation in detail. Due to the space available, we mainly describe the design related to the netgraph facility. The rest of the implementation is straightforward and does not really need much comment at the implementation level. In this section, all reported object code figures refer to code compiled in FreeBSD 4.5-STABLE on i386, using the supplied GNU `gcc` 2.95.3 tool chain.

4.1 EAPOL netgraph node

The EAPOL netgraph node is a new netgraph node that implements the EAPOL protocol as defined in IEEE 802.1x draft 11 [1]. However, since EAPOL includes a few layering violations, the EAPOL node is forced to have some rudimentary understanding of the protocol above, EAP, as well. That is, an EAPOL implementation is expected to send a few canned EAP packets (identity request, forced success, and forced failure) as well as understand the meaning of success and failure packets as received from an authentication server.

File name	Source lines	Object code size
ng_eapol.c	1038	3980 + 164
ng_eapol_base.c	255	1332 + 68
ng_eapol_auth.c	658	2756 + 508
ng_eapol_supp.c	552	2432 + 372
ng_eapol_macfilter.c	89	372 + 0
ng_eapol.h	77	
ng_eapol_base.h	326	
Total ng_eapol.ko	2995	19013 + 1220

Table 1: The EAPOL netgraph module

The implementation is divided into four source files (see Table 1). `ng_eapol.c` implements the netgraph related code, `ng_eapol_base.c` implements functionality common to both authenticator and supplicant, while `ng_eapol_auth.c` and `ng_eapol_supp.c` the role specific functionality. We also considered separating the authenticator and supplicant functionality into separate netgraph modules, but given that over one third of the code volume is spent on netgraph glue, we decided to integrate them into a single node instead. However, it is still possible to leave out either authenticator or supplicant functionality (or both) through compile-time options. This also simplifies testing.

```
handle_eapol(...,
    struct mbuf *m,
    meta_p meta) {
  struct eapol_hdr *ep = mtod(...);

  switch (ep->eapol_code) {
  case EAPOL_CODE_EAP:
    return handle_eap(.., m, meta);
  case EAPOL_CODE_LOGOFF:
    eapolp->eapol_state = LOGOFF;
    NG_FREE_M(m);
    NG_FREE_META(meta);
    return 0;
  ...
  }
}
```

Figure 7: Result of functional programming style

Another design choice we faced was between a functional vs. procedural programming style. In a more functional style, the program is divided into separate functions that have little if any side effects. Each piece of code performs memory management separately. In a pure functional programming language, such as Lisp or ML, memory management is taken care by the runtime, and side effects are virtually nonexistent. In a more procedural style, procedures act on data structures, and memory management centers around the them.

In our case, each netgraph node is separately responsible of either passing all packets received to another node, or explicitly freeing the mbuf and meta objects that comprise the packet. In the first version of the code, the actual code for these operations was distributed to the point of code where the decision was made. A simplified example is shown in Figure 7, above. This led to a situation where it was fairly hard to ensure that there was no memory leaks, since a single function was forced to free the mbuf and meta sometimes while sometimes delegating this to the called functions.

After realizing this, the code was refactored leading to a design where the packet is either forwarded or freed centrally at the very first function where the packet enters the node, `ng_eapol_rcvdata`. This makes sure that there are no memory leaks, but this also necessitates that, instead of passing mbufs, callers pass a pointer to the original mbuf pointer, and an output parameter that specifies a netgraph hook. If the returned hook is non-`NULL`, the packet is passed to the hook returned. The resulting code is depicted in Figure 8, on the next page.

In the refactored code, the metadata does not need to be passed around any more (unless it is used, or course),

```
handle_eapol(...,
    struct mbuf **mp,
    hook_p *nhook) {
  struct eapol_hdr *ep = mtod(...);

  switch (ep->eapol_code) {
  case EAPOL_CODE_EAP:
    return handle_eap(.., mp, nhook);
  case EAPOL_CODE_LOGOFF:
    eapolp->eapol_state = LOGOFF;
    *next_hook = NULL;
    return 0;
  ...
  }
}
```

Figure 8: Refactored code

and freeing the packet is replaced by setting the next hook output parameter `nhook` to `NULL`, thereby signalling that the packet should be freed and not passed on. Passing `mbuf **` instead of `mbuf *` is necessitated by some mbuf routines explicitly freeing the mbuf in the case of mbuf shortage. By passing a pointer to the mbuf pointer, we avoid freeing mbufs twice in those cases.

Thus, the basic lesson learned was that the functional programming style that the author is used to use with languages that include garbage collection just does not work when you have to take care of memory management yourself. It is much better to centralize the actual code that manages memory, and use extra parameters to signal the decision of what to do from the actual decision point to the centralized piece of code.

Other than the issues with granularity and programming style, implementing the EAPOL node was pretty straightforward translation of the standard specification into working code.

4.2 MAC filter netgraph node

The MAC filter node, `ng_macfilter`, is a fairly simple mux/demux. It is build around an ordered table of MAC addresses. Each address in the table is annotated with the index of an upper netgraph hook. All packets received from any of the upper hooks are passed directly to the lower hook. On the other hand, whenever a packet is received from the lower hook, the source MAC address is inspected to see if it matches with any of the addresses in the table. If a match is found, the packet is passed to the indicated upper hook. If no match is found, the packet is passed to the default upper hook.

File name	Source lines	Object code size
ng_macfilter.c	792	2032 + 100
ng_macfilter.h	60	
Total ng_macfilter.ko	852	4110 + 200

Table 2: The macfilter module

4.3 User level modules

To test the netgraph modules and to implement one of our usage scenarios, we implemented EAP OTP authenticator and supplicant modules. These modules both work on the user level, and rely on a small new library, `libeap`. They also utilize the `libskey` library present in FreeBSD.

File name	Source lines	Object code size
libeap/eap_auth.c	144	1020 + 0 + 0
libeap/eap_peer.c	165	788 + 0 + 0
eap-opie/opie.h	25	
eap-opie/opie_calc.c	108	436 + 48 + 228
eap-opie/opie_auth.c	149	940 + 0 + 1536
eap-opie/opie_peer.c	112	764 + 0 + 1536
opie_auth *binary*	10212 bytes	5125 + 380 + 2348
opie_peer *binary*	9575 bytes	4553 + 368 + 2116
Total source code	703 lines	

Table 3: The user level modules

As Table 3 shows, the actual sizes of the user level programs, as reported by `size(1)`, are extremely small.

4.4 Overall experiences with netgraph

This project was the first time when the author used netgraph, even though he had some prior experience in working with kernel level code. In general, netgraph turned out to be a very well designed facility that boosted the project productivity considerably. The main benefit was the no-hassle startup, and the ability to write all the code as loadable kernel modules. That is, once the author decided to use netgraph, the first null version of the to-be EAPOL module was loaded into the kernel within a couple of hours, most time spent on reading netgraph documentation. The documentation was clean and concise; the only part that the author found lacking was the usage examples, which were somewhat hard to understand. That led to some unnecessary work before

really understanding how to use the `ngctl` program and how the generic netgraph command messages such as `mkpeer` or `connect` work. In general, it looks like the netgraph facility provides an extremely good environment for implementing and connecting together sub-IP protocols in FreeBSD.

5 Usage scenarios

In this section, we describe three different advanced usage scenarios. These all go beyond those intended to be implemented by the base IEEE 802.1x standard, which is mainly meant to protect corporate intranets from outsiders. In the first scenario, we outline a community membership based authorization model, where members of a community may use the network regardless of their identity. In the second scenario we outline how to implement a pre-paid, pay-per-use scenario suitable for early public WLAN adopters at airports and hotels. Finally, in the last scenario we outline how to introduce multioperator support into a WLAN access network.

The different scenarios have different threat and trust models. We discuss the security needs of the scenarios only briefly; more thorough treatment of the underlying trust models is beyond the scope of this paper.

None of these scenarios have been fully implemented. The second, pre-paid, pay-per use scenario is close to being really implemented, but even there are bits and pieces that are missing before it could be used in real life. The scenarios are provided to exemplify how the flexibility of the implementation allows it to be used as a building block in fulfilling different needs.

5.1 Community membership

There are number of open WLAN communities, e.g. Seattle Wireless [8] or Electrosmog [9], where the members of the communities are basically providing free WLAN based Internet access to anybody passing by. While these schemes work fine today, they are likely to suffer from "the tragedy of the commons" phenomenon as the number of WLAN users starts to grow. One possible way to continue the spirit of these networks while protecting them from overuse is to limit their usage to members, i.e., to create closed communities.

Using 802.1x, EAP TLS [6], and certificates, it seems to be fairly easy to create closed but decentralized communities. Let us assume that there are a number of founding members that create the community, and that any two of the founding members can accept new members to the community. One way of implementing such a scheme is to use authorization certificates [10], and to let all the founding members to sign a policy certificate stating that any two members are eligible to introduce new members. Thus, instead of having a single centralized certification authority (CA) the community would distribute the responsibility among the founding members.

All community access points would be connected to a centralized repository that contains the initial policy certificates. When new users come to an access point, their laptop would run EAP TLS and send the two certificates signed by the founding members to the access point. The access point would fetch the policy certificates corresponding to the signers of the user certificates, and make the access decision normally using the KeyNote2 engine.

In this way, the community would limit access to its members, while still retaining the decentralized nature of deciding who is a member and who is not. This solution requires that KeyNote2 [10] is integrated to OpenSSL [11], and that EAP TLS is created using the OpenSSL library.

Compared to NoCatAuth [12][13], probably the best known open source effort on this area, this 802.1x based solution would be better in two ways. Firstly, the 802.1x allows the authentication to happen without any user intervention. Secondly, the use of authorization certificates allows the administration of group membership to be completely decentralized.

5.2 Pre-paid, pay-per-use

As another scenario, we show how OTP [7] accounts can be used as pre-paid pay-per-use value tokens, and how it is possible to easily add a captive portal to the access point. Integrating these with some kind of on-line macropayment system, such as credit card payment, it is possible to create a pre-paid, pay-per-use public WLAN that is open to any users.

The first key element in this scenario is to realize the potential of OTP accounts and re-authentication. That is, the periodic re-authentication feature of 802.1x allows the access point to request the next OTP token from the user. Thus, we can establish a convention where a single OTP token represents the value for a certain commodity, e.g., for one minute of access time. Once the commodity has been used up, the access point can request for the next token in order to continue the service.

In this way, an OTP account can be used to represent a pre-paid, pay-per-use account. The user buys such an account with a credit card or some other means, and gets

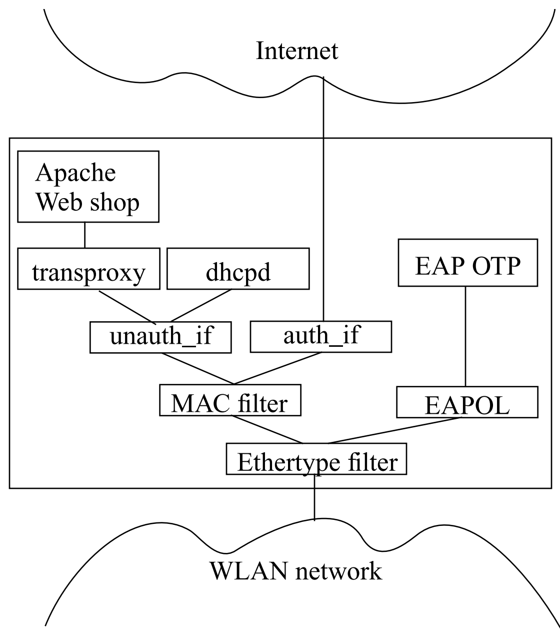

Figure 9: Pre-paid, pay-per-use access point

the account name and the OTP seed value. In most cases there is no value in requesting a password from the user, and therefore it is possible to automate the whole purchase process. For example, a small web browser plug-in can be used to transfer the account from the shop to a preset location at the user's hard disk. The EAP OTP supplicant is then able to select the next unused account from this location.

The final step is to make it easy to buy new OTP accounts. It must also be easy to become a user, i.e. to buy the initial OTP account and, if necessary, download the 802.1x and EAP OTP implementations. This can be easily accomplished with a captive portal. As we described in Section 3.4, our MAC filter module allows packets arriving from unauthenticated users to be passed on instead of being dropped. The netgraph facility allows these packets to be passed to an pseudo-interface, which can then pass them to IP. Using the FreeBSD IP firewall, ipfw, it is then easy to trap HTTP all packets arriving from the specific pseudo-interface, and use a forwarding rule (`fwd`) to pass them to a transparent web proxy such as transproxy [14]. The transparent proxy can then tunnel the packets to a web server, which can be located either at the access point itself or somewhere else.

Thus, with suitable configuration it is possible to create a situation where the only services provided to an unauthenticated client are DHCP and the captive portal. This effectively leads to a situation where the client is served an IP address, but no matter which web site the users attempt to access, they will be redirected to the captive portal. The captive portal web site, in turn, provides the possibility to buy OTP accounts and to download all necessary software.

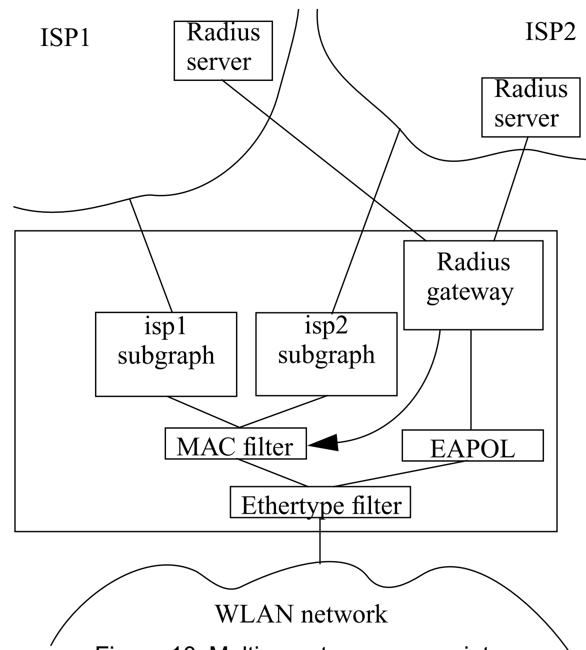

Figure 10: Multioperator access point

5.3 Multioperator support

As a final usage example, we show how the extensibility of our implementation can be used to support a multi-operator scenario where a single WLAN network is shared between multiple ISPs and other service providers. In this case, we need a slightly more intelligent EAP authenticator. This authenticator needs to be able to talk to multiple back-end authentication servers — RADIUS seems to be fine for that — and, more importantly, to command the MAC filter to connect the clients to different upstream netgraph hooks depending on the commands received from the authentication servers. A possible setup is depicted in Figure 10.

The upstream hooks are all individually connected to a subgraph that transports the packets to the ISP. In a very simple (but unrealistic) example, each ISP would have a separate Ethernet interface in the access point. In that case, the hooks are simply connected to the Ethernet interface, and the access point just bridges the packets, selecting the outgoing interface based on the source MAC address. In a more realistic case, there would be some netgraph subgraph that tunnels the packets to the ISP, using e.g. 802.1q VLAN or PPPoE. Netgraph allows one to build such subgraphs fairly easily. As a result, all the data packets would be handled inside the kernel, potentially leading to fairly good performance.

6 Performance measurements

To get a feeling about the real life performance of the technology and implementation, we implemented a limited pay-per-use WLAN scenario. In the setting we assumed that the laptop user already has purchased the necessary tokens (the OTP account) and is merely connecting to a network where the tokens can be used. The setting is illustrated in Figure 11, below. In the performance test, we used an old 400 MHz PII 64 Mb laptop, with Lucent WaveLAN silver card, an Apple Airport (graphite) 802.11b basestation, and an old 350 MHz Celeron 128 Mb desktop PC. The Apple Airport was configured to be a transparent bridge.

Using the given configuration, we measured connection setup time and throughput, both without and with our 802.1x implementation. The results are given in Table 4. The connection setup measures the time, in seconds, from inserting the Lucent WLAN card to the laptop to the time when the laptop received the IP address via DHCP. The bulk transfer throughput measures the additional overhead caused by the MAC filter in the kernel, using FTP file transfer as the measurement tool. The values are averages over five measurements.

Measurement	Without 802.1x	With 802.1x
Connection setup	22 ± 2 s	24 ± 3 s
Bulk transfer throughput	830 KB/s	820 KB/s

Table 4: Performance results

The results show that the overhead is negligible. The connection setup takes 2–3 seconds longer, an increase of roughly 10%. This is consistent with our other measurements that showed the 802.1x exchange itself taking 1.1 ± 0.4 seconds on the average. The performance tests gave slightly better performance when using 802.1x, possibly due to interactions with 802.3 or 802.11b queuing and retransmission algorithms.

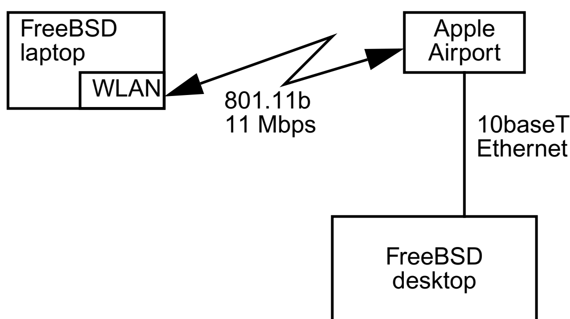

Figure 11: Performance Measurement Setting

7 Open issues and future work

7.1 Security shortcomings

Many people, including Mishra and Arbaugh [15], have argued that 802.1x security is flawed since it does not provide per-packet integrity and authenticity. Depending on setting, that may allow session hijacking, basically allowing an attacker to take over a MAC address that belongs to a legitimate and authenticated user.

We completely agree with the basic reasoning. To be properly secure in a shared medium environment, such as 802.11 WLAN, 802.1x authentication should be tightly integrated with a link-level integrity system that would use different session keys for different clients. However, today there are no standards for such link-level encryption. Besides, in some of the given scenarios, such as the pre-paid pay-per-use, the benefits a session hijacker might get from its attack may be questioned. However, the potential benefit depends on the specific characteristics of the underlying link-layer; if the link-layer makes it impossible for two clients to use the same MAC address at the same time, the attacker would have difficulties to use more than one pre-paid time slot.

If we really want to secure the wireless link against intruders, there are a few possible approaches. One would be to develop further the proposed IEEE Robust Security Network (RSN) architecture along the lines suggested in [15]. That would involve using the Wireline Equivalent Privacy (WEP) enhanced with changing keys derived from keying material received via 802.1x. Another possibility would be to use IPsec Authentication Header (AH) between the client and the access point, deriving the AH session keys from the keying material provided by e.g. TLS over EAP. Still a different approach would be to use IPsec IKE instead of 802.1x in authenticating the clients, and then use AH. However, all such approaches are beyond the scope of this paper.

Thus, within the scope of this work we have limited ourselves to checking the MAC addresses. We acknowledge that this is not always sufficient and definitely not the right long term solution. However, for our main scenario, pre-paid pay-per-use, the achieved security level seems appropriate for the time being. The only additional security feature that we are considering to add in the future is to combine MAC level and IP level filtering. That blocks simpler attacks where the attacker just sends packets using the same MAC address as some other node but a different IP address.

7.2 From OTP to micropayments

From the open source and grassroots movement point of view, the most lucrative usage scenario would be one where the good properties of the closed community scenario and the pre-paid pay-per-use scenario would be combined. However, it is not at all clear how such a scheme could be achieved. For example, we have considered to putting a full-fledged three-party micropayment protocol over the top of EAP, and passing micropayment tokens from the client to the access point. That would allow the client to pay to the access point, and the access point owner would then be able to use the tokens collected by the access point somewhere else. Unfortunately there seems to be problems in the trust structure, and more research is needed to figure out how such a structure and decentralized administration could be combined.

8 Conclusions

In this paper, we have shown that the IEEE 802.1x standard can be used to solve useful problems beyond its original purpose. Furthermore, we have shown that the FreeBSD netgraph facility makes it very easy to implement the protocol, and at the same time to provide a flexible architecture that allows the same software modules to be applied to widely varying purposes. In particular, we have illustrated that, in addition to user authentication, unmodified 802.1x can be used for pre-paid, pay-per-use access. We have also shown that is fairly easy to support multiple operators at a single Wireless LAN using the FreeBSD netgraph facility.

Acknowledgments

I am indebted to David Partain and Per Magnusson of Ericsson Research, who worked for a few months with me at the iPoints innovation cell. Many of the central ideas for this paper were developed together with them during that time. Thus, whenever I am referring to collective effort in this paper, I am usually referring to the work conducted together with David and Per. However, I alone am responsible for any mistakes and shortcomings in this paper.

I also want to thank Juha Heinänen of Song Networks for pointing out the importance of 802.1x in the first place, and especially for his insights and ideas related to the multioperator scenario. I am also thankful to Julian Elischer for his help in implementing the netgraph modules, and to Craig Metz and Marco Moteni for their comments on various versions of the paper.

Availability

An alpha version of the EAPOL code is available at `http://www.tml.hut.fi/~pnr/eapol/`. The distribution includes the netgraph modules and simple EAP OTP authenticator and supplicant user level programs, together with some scripts used for testing and performance measurements.

References

[1] *IEEE Draft P802.1X/D11: Standard for Port based Network Acce*ss Control, LAN MAN Standards Committee of the IEEE Computer Society, March 27, 2001.

[2] L. Blunk and J. Vollbrecht, *RFC2284, PPP Extensible Authentication Protocol (EAP),* IETF, March 1998.

[3] C. Rigney, S. Willens, A. Rubens, W. Simpson, RFC2865, Remote Authentication Dial In User *Service (RADIUS)*, IETF, June 2000.

[4] Archie Cobbs, *All about netgraph*, daemon news, March 2000, http://www.daemonnews.org/200003/netgraph.html

[5] Nick Petroni, Bryan D. Payne, Bill Arbaugh, and Arunesh Mishra, *Open1x project home page,* University of Maryland, February 2002, `http://www.open1x.org`,

[6] B. Aboba, D. Simon, *RFC2716, PPP EAP TLS Authentication Protocol,* IETF, October 1999.

[7] N. Haller, C. Metz, P. Nesser, M. Straw, *RFC2289, A One-Time Password System,* Internet Engineering Task Force, February 1998.

[8] Seattle Wireless Home Page, http://seattlewireless.net/

[9] Electrosmog Home Page, http://elektrosmog.nu/

[10] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis, *RFC2704, The KeyNote Trust-Management System Version 2,* Internet Engineering Task Force, September 1999.

[11] OpenSSL Home page, http://www.openssl.org

[12] NoCatAuth White Paper, http://nocat.net/nocatrfc.txt

[13] Rob Flickenger, *NoCatAuth: Authentication for Wireless Networks,* O'Reilly Network Wireless Devcenter, November 9th 2001, http://www.oreillynet.com/pub/a/wireless/2001/11/09/nocatauth.html

[14] John Saunders, *transproxy*, ftp://ftp.nlc.net.au/pub/unix/transproxy/

[15] Arunesh Mishra and William A. Arbaugh, "An Initial Security Analysis of the IEEE 802.1X Standard", UMIACS-TR-2002-10, University of Maryland, February 2002.