Piszemy własny moduł PAM

Andrzej Nowak

Na CD:

Na dołączonej do pisma płycie CD zamieszczone są wykorzystywane w artykule programy i dokumentacja.

Artykuł jest skierowany przede wszystkim do administratorów systemów linuksowych, którzy chcą wprowadzić własne zmiany do swoich systemów zabezpieczeń, ale również do tych, którzy chcą dowiedzieć się czegoś o PAM od kuchni.

Umiejętność pisania modułów PAM jest bardzo przydatna, gdy tworzymy niestandardowy lub nietypowy system uwierzytelniania. Ponadto przekazywana tu wiedza z pewnością przyda się przy modyfikacjach konfiguracji PAM wykraczających poza tematy zawarte na ubogich nieraz stronach manuali. Analiza kodu źródłowego modułu może często wyjaśnić wiele pozornych nieprawidłowości w jego funkcjonowaniu albo nietypowych zachowań.

Przedstawiam wprowadzenie do API systemu PAM na podstawowym poziomie, ale artykuł wymaga od Czytelnika znajomości języka C na poziomie średnim – tzn. takim, który umożliwi mu zrozumienie przedstawionych kawałków kodu oraz ewentualnie napisanie własnych. Atutem jest każdy miesiąc doświadczenia linuksowego, które Czytelnik posiada. Trzeba to powiedzieć wprost: zabawa z PAM nie zawsze jest łatwa i przyjemna. Jednak satysfakcja i korzyść, która płynie z przezwyciężenia PAM-owych trudności, jest – zapewniam Was – niebagatelna.

Co to jest PAM?

Na początek parę słów do tych, którzy dopiero zaczynają poznawać zagadnienie. PAM to skrót od angielskiego *Pluggable Authentication Modules*. Jest to zestaw bibliotek oraz interfejs programistyczny dla systemów uniksowych i linuksowych, umożliwiający administratorowi dokładny wybór sposobu weryfikacji dostępu do usług. Mechanizm ten jest integralną częścią wielu współczesnych dystrybucji Linuksa (np. Red Hat, Debian), a także BSD (FreeBSD od wersji 3.1).

Idea PAM powstała w firmie Sun Microsystems i bardzo szybko chwyciła. Pierwotnie w Solarisie PAM występował w postaci niejawnej, tzn. nie było pliku konfiguracyjnego ani stosu modułów. Obecnie jego zachowanie można kontrolować poprzez odpowiednie wpisy w */etc/pam.conf*. Z innych znanych uniksów PAM występuje np. w HP-UX (pojawił się w wersji 11). W tym tekście będę koncentrował się na tzw. Linux-PAM, czyli implementacji PAM dla Linuksa – pisząc *PAM* będę miał na myśli *Linux-PAM*.

Ogólne założenia

Ogromna ilość aplikacji i usług uwzględnia i wykorzystuje istnienie PAM – są to tzw. *pam-aware applications* – dzięki czemu zarządzanie dostępem do nich staje się relatywnie łatwe. Nie potrzeba ponownej kompilacji programu, aby w dowolny sposób zmienić tryb dostępu do niego. Oto wybrane podstawowe założenia systemu PAM, przedstawione w dokumencie RFC #86:

- administrator powinien mieć możliwość wyboru domyślnego mechanizmu autentykacji dla danego systemu,
- konfiguracja autentykacji musi być związana z konkretną usługą (czyli każda usługa ma swój zestaw reguł dotyczących uwierzytelniania i nadawania praw),
- istnieje możliwość konfiguracji więcej niż jednego sposobu autentykacji dla danej aplikacji,
- moduły układa się w stos, aby użytkownik nie musiał kilka razy podawać tego samego hasła, ale z drugiej strony istnieje możliwość podawania różnych haseł podczas próby dostępu do jednej usługi,
- aplikacje *pam-aware* są niezależne od zmian w PAM, tzn. nie jest wymagana ich rekompilacja aby zmienić sposób dostępu.

Jak działa PAM?

Są cztery niezależne dziedziny, w których operuje PAM:

- zarządzanie autentykacją (ang. *authentication management*),
- zarządzanie kontem (ang. *account management*),
- zarządzanie sesją (ang. *session management*),
- zarządzanie hasłem (ang. *password management*).

Autor jest studentem II roku informatyki na Politechnice Gdańskiej. Od sześciu lat interesuje się systemami Unix/Linux, a w szczególności problematyką zabezpieczeń systemów komputerowych. Kontakt z autorem: andrzej_nowak@o2.pl

Obsługa autentykacji (authentication management)

Aby poprawnie zainicjalizować możliwości zarządzania autentykacją we własnym module, dyrektywę `#include <security/pam_modules.h>` należy poprzedzić definicją stałej `PAM_SM_AUTH` (`#define PAM_SM_AUTH`)

Funkcja `pam_sm_authenticate()` może zwrócić następujące wartości:

- `PAM_SUCCESS` – proces autentykacji powiódł się i wszystko jest w porządku,
- `PAM_AUTH_ERR` – nie udało się pomyślnie przeprowadzić autentykacji,
- `PAM_USER_UNKNOWN` – moduł nie zna takiego użytkownika,
- `PAM_AUTHINFO_UNAVAIL` – zawiodła usługa zapewniająca dostęp do informacji,
- `PAM_CRED_INSUFFICIENT` – aplikacja nie ma wystarczających uprawnień aby przeprowadzić proces uwierzytelnienia,
- `PAM_MAXTRIES` – jeden lub więcej modułów autentykacji wyczerpał maksymalną ilość prób.

Funkcję `pam_sm_authenticate()` można wywołać z flagą `PAM_DISALLOW_NULL_AUTHOK`. Jej użycie powinno spowodować, że moduł w przypadku natrafienia na pusty token zwróci wartość `PAM_AUTH_ERR`. W domyślnym przypadku jednak moduł powinien zezwalać na użycie pustego tokena (np. pustego hasła).

Flagi funkcji `pam_sm_authenticate()`:

- `PAM_SILENT`,
- `PAM_DISALLOW_NULL_AUTHTOK` – moduł powinien zwrócić `PAM_AUTH_ERR`, jeśli token użytkownika jest pusty; bez tej flagi użytkownik w wymienionym przypadku nie będzie pytany o hasło.

Funkcja `pam_sm_setcred()` może zwrócić:

- `PAM_SUCCESS`,
- `PAM_CRED_UNAVAIL` – nie udało się ustalić praw użytkownika,
- `PAM_CRED_EXPIRED` – upłynął termin ważności praw użytkownika,
- `PAM_USER_UKNOWN` – moduł nie zna takiego użytkownika,
- `PAM_CRED_ERR` – nie udało się ustawić praw użytkownika.

Flagi funkcji `pam_sm_setcred()`:

- `PAM_ESTABLISH_CRED` – ustaw uprawnienia związane z usługą uwierzytelniającą,
- `PAM_DELETE_CRED` – usuń uprawnienia związane z usługą uwierzytelniającą,
- `PAM_REINITIALIZE_CRED` – odnów uprawnienia użytkownika,
- `PAM_REFRESH_CRED` – przedłuż uprawnienia użytkownika (w czasie).

Podczas autentykacji ma miejsce pobranie hasła (ogólnie: tokena) od użytkownika i sprawdzenie jego poprawności, a także nadanie użytkownikowi uprawnień.

Jeśli podane hasło jest dobre (ogólnie: jeśli PAM uzna, że użytkownik jest tym, za kogo się podaje), rozpoczyna się proces, w którym PAM sprawdza, czy użytkownik ma dostęp do konta (*account management*). Takie problemy jak wygaśnięcie hasła albo wyczerpana ilość loginów są zgłaszane właśnie na tym etapie.

Później następuje przejście do zarządzania sesją. PAM jest obecne zarówno przy otwieraniu sesji, jak i przy jej zamykaniu. Ogólnie moduły znajdujące się w grupie zarządzania sesją przeprowadzają wszelkie czynności niezbędne do poprawnego zalogowania się użytkownika – na przykład montują dyski, przygotowują usługi.

Ostatnia grupa, zarządzanie hasłem, jest potrzebna tylko w momencie próby zmiany uprawnień dostępu (np. haseł) przez użytkownika.

Na konfigurację PAM składają się plik */etc/pam.conf* oraz pliki w katalogu */etc/pam.d/*. Zawierają one nazwy usług lub programów, a w każdym z nich zawarty jest zestaw reguł autentykacji dla danej aplikacji. Zawartość katalogu */etc/pam.d/* i jego plików może się różnić nawet znacznie między poszczególnymi dystrybucjami Linuksa, dlatego nie będę omawiał jego struktury – zachęcam do pozwiedzania na własną rękę. Zauważmy, że nie ma gwarancji, że konfiguracja stosu modułów przekopiowana z jednej dystrybucji na inną będzie działać. Co więcej, należy spodziewać się po takiej operacji niemałych problemów.

Przy okazji chciałbym przypomnieć o parametrze jądra, który można podać przy bootowaniu Linuksa: `init=/bin/ash.static` (lub inny statyczny shell). Parametr ten pozwala zabootować system z ominięciem PAM, z pewnością przyda się w sytuacji, kiedy PAM ma popsutą konfigurację. Przed przeprowadzeniem bardziej skomplikowanych modyfikacji warto utworzyć kopie zapasowe plików konfiguracyjnych.

Jak powinien wyglądać moduł?

Z racji faktu, że istnieje ogólny interfejs API dla PAM, każdy moduł powinien bazować na pewnym ogólnym szkielecie. Budując program zgodnie z powszechnymi zaleceniami zwiększamy jego niezawodność w sytuacjach kryzysowych – np. przy braku dostępu do sieci. Zachowanie się modułu w trudnych warunkach może mieć kluczowe znaczenie dla poprawności działania całego systemu.

Przede wszystkim należy pamiętać o niezależnej obsłudze każdej z czterech grup zarządzania (autentykacja, konto, sesja i hasło). Użytkownik może wywoływać obsługę kolejnych grup w dowolnej kolejności, więc nie należy polegać na uprzednim poprawnym wykonaniu instrukcji z innej części kodu niż ta, do której aktualnie odwołuje się system. Jednak w pewnych sytuacjach możemy przyjąć założenia dotyczące wcześniej przeprowadzonych czynności: np. jeżeli wywoływana jest funkcja `pam_sm_open_session()`, to użytkownik został wcześniej uwierzytelniony. Moduł powinien zawierać

Obsługa konta (account management)

Aby poprawnie zainicjalizować możliwości zarządzania kontem we własnym module, dyrektywę `#include <security/pam_modules.h>` należy poprzedzić definicją stałej `PAM_SM_ACCOUNT` (`#define PAM_SM_ACCOUNT`)

Funkcja `pam_sm_acct_mgmt()` może zwrócić następujące wartości:

- `PAM_SUCCESS`,
- `PAM_ACCT_EXPIRED` – konto wygasło i użytkownik nie może się już logować,
- `PAM_AUTH_ERR` – wystąpił błąd autentykacji,
- `PAM_AUTHTOKEN_REQD` – hasło wygasło; użytkownik prawdopodobnie zostanie zapytany o nowe,
- `PAM_USER_UNKNOWN` – moduł nie zna takiego użytkownika.

Flagi: jak dla funkcji `pam_sm_authenticate()`.

taki podzbiór sześciu podstawowych funkcji (przedstawiam je nieco dalej), aby był zdolny do działania przynajmniej w jednej grupie.

Kolejnym istotnym zaleceniem jest poprawna obsługa wywołań funkcji – spośród wspomnianych wcześniej sześciu – których ciało nie zostanie zaimplementowane. Twórcy PAM zalecają, aby w zależności od potrzeby funkcje takie zwracały wartość `PAM_SUCCESS`, `PAM_SERVICE_ERR` lub `PAM_IGNORE` (stałe `PAM_*` oraz ich znaczenia znajdują się w Ramkach *obsługa autentykacji*, *obsługa konta, obsługa sesji*, *obsługa hasła*).

W dalszej perspektywie należy uwzględnić fakt, że z pliku konfiguracyjnego mogą zostać przekazane jakieś argumenty, np. *debug*. Możemy je odczytać w tradycyjny sposób, używając zmiennych `int argc` i `char *argv[]` jako parametrów. Zwracam uwagę, że zmienna `argv[0]` nie zawiera nazwy modułu, jak to ma miejsce w zwykłych programach! Jest ona wskaźnikiem na początek pierwszego argumentu.

Programiści powinni także zwrócić uwagę na szereg innych zagadnień. Przeważnie moduły PAM są ładowane dynamicznie – nie należy zatem używać zmiennych typu `static`. Zaleca się szerokie wykorzystanie licznych kodów powrotu (na Listingach 1-4).

Wchodzimy w głąb – podstawowe funkcje

Oto prototypy sześciu podstawowych funkcji, będących podstawowymi częściami składowymi modułów PAM:

- Autentykacja:

```
PAM_EXTERN int pam_sm_authenticate
  (pam_handle_t *pamh, int flags,
   int argc, const char **argv);
PAM_EXTERN int pam_sm_setcred
  (pam_handle_t *pamh, int flags,
   int argc, const char **argv);
```

- Zarządzanie kontem:

```
PAM_EXTERN int pam_sm_acct_mgmt
  (pam_handle_t *pamh, int flags,
   int argc, const char **argv);
```

- Zarządzanie sesją:

```
PAM_EXTERN int pam_sm_open_session
  (pam_handle_t *pamh, int flags,
   int argc, const char **argv);
PAM_EXTERN int pam_sm_close_session
  (pam_handle_t *pamh, int flags,
   int argc, const char **argv);
```

- Zarządzanie hasłem:

```
PAM_EXTERN int pam_sm_chauthtok
  (pam_handle_t *pamh, int flags,
   int argc, const char **argv);
```

Czytelnik z pewnością już zauważył, że do każdej funkcji przekazywane są takie same zestawy parametrów. Jest to rozwiązanie służące ujednoliceniu interfejsu modułu. Niezależnie od funkcji, jaką będzie pełnił (np. zarządzanie kontem czy hasłem), jego konfiguracja będzie przeprowadzana w identyczny sposób. Pierwszy parametr (`*pamh`) jest wskaźnikiem na uchwyt, przekazywanym do funkcji przez PAM. Drugi (`flags`) to flagi, z jakimi jest wywoływany moduł. Nie są to zwykłe parametry – te są przekazywane w dwóch następnych zmiennych. `Argc` oznacza ilość argumentów, `**argv` jest ich tablicą.

Jak widać obsługą autentykacji (*authentication management*) zajmują się dwie funkcje: `pam_sm_authenticate()` i `pam_sm_setcred()`. Do obu (jak do wszystkich omawianych

Obsługa sesji (session management)

Aby poprawnie zainicjalizować możliwości zarządzania sesją we własnym module, dyrektywę `#include <security/pam_modules.h>` należy poprzedzić definicją stałej `PAM_SM_SESSION` (`#define PAM_SM_SESSION`)

Funkcje `pam_sm_open_session()` oraz `pam_sm_close_session()` mogą zwrócić następujące wartości:

- `PAM_SUCCESS`,
- `PAM_SESSION_ERR` – błąd przy otwieraniu lub zamykaniu sesji.

Flagi:

- PAM_SILENT

funkcji) można dodać flagi przekazywane przez zmienną `flags`. Odpowiednie flagi dla poszczególnych funkcji zawarte są w Ramkach *obsługa autentykacji*, obsługa *konta, obsługa sesji*, *obsługa hasła*. Pierwsza z funkcji przeprowadza bezpośrednie czynności związane z autentykacją, druga zajmuje się ustalaniem i przygotowywaniem uprawnień użytkownika. Aplikacje powinny wywoływać ją po przeprowadzeniu uwierzytelnienia, ale przed rozpoczęciem sesji.

Autentykacja może być przeprowadzana na wiele różnych sposobów – programiści mają tutaj ogromne pole do popisu. Jeśli ktoś ma chęć i możliwości, może napisać kod obsługujący czytnik linii papilarnych albo skaner siatkówki oka i używać takiego modułu do uwierzytelniania użytkownika przy komputerze z podłączonym urządzeniem. Bardziej prozaicznym pomysłem jest funkcja, która żąda od *roota* oprócz zwykłego hasła także hasła dnia, np. na stałe wkompilowanego w moduł – w ten sposób potencjalny włamywacz miałby małą niespodziankę.

Kawałek kodu zajmujący się weryfikacją dostępu do konta (*account management*, `pam_sm_acct_mgmt()`) w ogólności powinien umożliwić ustalenie, czy użytkownik ma w danej chwili zezwolenie na wstęp. Można się spodziewać, że użytkownik przeszedł wcześniej uwierzytelnianie. `Pam_sm_acct_mgmt()` jest dobrym miejscem, od którego można zacząć pisanie własnych procedur w module. Dla przykładu, można w tym miejscu zamieścić funkcję, która sprawdza dzień tygodnia. Jeśli jest piątek, nie wpuszcza do systemu użytkowników próbujących zalogować się przy użyciu klientów SSH spod Windows (ten złośliwy pomysł nie jest świeży, bowiem istnieje już moduł do serwera Apache, który w piątki odmawia współpracy z przeglądarkami Internet Explorer). Z praktyczniejszych pomysłów: można pokusić się o umieszczenie wewnątrz omawianej funkcji kod wpuszczający do systemu określonych użytkowników o określonych godzinach – podobny przykład rozważymy dalej.

Zarządzanie sesją to część odpowiedzialna za czynności, które trzeba wykonać bezpośrednio przed otwarciem dostępu do usługi i po jego zamknięciu. Przykłady: logowanie informacji o otwarciu sesji, montowanie i odmontowanie katalogów, wyświetlanie komunikatów, przygotowywanie danych etc.

Funkcja `pam_sm_chauthtok` jest wywoływana, gdy moduł zostanie dołączony do stosu grupy *password* (*Password Management*). Użytkownik chcący zmienić hasło będzie musiał przebić się przez niespodzianki, które mu przygotujemy. Dla przykładu, możliwości modułu `pam_cracklib` mogą okazać się dla nas niewystarczające, bo z bliżej nieokreślonego powodu chcemy, aby każdy użytkownik posiadał w haśle przynajmniej trzy znaki `$`. Wtedy odpowiednio napisana funkcja `pam_sm_chauthok` umieszczona we własnym module PAM pomoże ten nakaz wyegzekwować.

Argumenty do modułu

Moduł powinien (ale nie musi) obsługiwać grupę standardowych argumentów, które może przekazać użytkownik. Grupa ta jest zdefiniowana w dokumentacji do PAM (*Linux-PAM Module Writers' Guide*).

- *debug* – po otrzymaniu tego argumentu moduł powinien wyrzucać do logów większe ilości informacji niż normalnie (*debug information*),
- *try_first_pass* – stosowane przy modułach *passwd* i *auth*; w przypadku otrzymania takiego argumentu, moduł powinien spróbować skorzystać z hasła przekazanego przez wcześniejszy moduł, a jeśli to się nie powiedzie, spytać o swoje hasło,
- *use_first_pass* – jak wyżej, z jedną różnicą: gdy próba uwierzytelnienia z otrzymanym hasłem nie powiedzie się, moduł powinien zakończyć ją z negatywnym wynikiem,
- *expose_account* – użycie tego argumentu oznacza, że administrator nie ma nic przeciwko ujawnianiu informacji

Obsługa hasła (password management)

Aby poprawnie zainicjalizować możliwości zarządzania hasłem we własnym module, dyrektywę `#include <security/pam_modules.h>` należy poprzedzić definicją stałej `PAM_SM_PASSWORD` (`#define PAM_SM_PASSWORD`).

Funkcja `pam_sm_chauthtok()` może zwrócić następujące wartości:

- `PAM_SUCCESS`,
- `PAM_AUTHTOK_ERR` – nie udało się uzyskać nowego hasła,
- `PAM_AUTHTOK_RECOVERY_ERR` – nie udało się uzyskać starego hasła,
- `PAM_AUTHTOK_LOCK_BUSY` – hasło jest zablokowane,
- `PAM_AUTHTOK_DISABLE_AGING` – starzenie się hasła zostało zablokowane,
- `PAM_PERM_DENIED` – odmowa dostępu,
- `PAM_TRY_AGAIN` – przygotowania do zmiany hasła nie powiodły się,
- `PAM_USER_UNKNOWN` – moduł nie zna takiego użytkownika.

Flagi:

- `PAM_CHANGE_EXPIRED_AUTHTOK` – oznacza, że hasło powinno być zmieniane tylko w przypadku wygaśnięcia; ta flaga musi występować w połączeniu z dwiema następnymi,
- `PAM_PRELIM_CHECK` – sprawdzanie gotowości modułu do zmiany hasła użytkownika; jeśli moduł nie jest gotowy, powinien zwrócić `PAM_TRY_AGAIN`,
- `PAM_UPDATE_AUTHTOK` – moduł powinien zmienić hasło użytkownika w tym wywołaniu funkcji `pam_sm_chauthtok()`.

Uwaga: funkcja `pam_sm_chatuthtok()` jest wywoływana dwa razy: najpierw z flagą `PAM_PRELIM_CHECK`, a następnie (jeśli moduł nie zwróci błędu `PAM_TRY_AGAIN`) z flagą `PAM_UPDATE_AUTHTOK`.

Listing 1. *pam_czas.c*

```
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#define PAM_SM_ACCOUNT
#include <security/pam_modules.h>
#include <security/_pam_macros.h>

PAM_EXTERN int pam_sm_acct_mgmt(pam_handle_t *pamh,
   int flags, int argc, const char **argv) {
int odpowiedz = PAM_AUTH_ERR, ret = PAM_AUTH_ERR;
const char *uzytkownik = NULL, *tempptr = NULL;
char *info = "Nie mozesz sie teraz zalogowac.\n";
struct pam_conv *rozmowa;
struct pam_message komunikat
struct pam_message *pkomunikat = &komunikat;
struct pam_response *resp = NULL;
time_t sekundy;
struct tm czas;

   ret = pam_get_user(pamh, &uzytkownik, NULL);
      if(ret != PAM_SUCCESS) return ret;

   if(!strncmp(uzytkownik,"root",4))
      odpowiedz = PAM_SUCCESS;
   else {
      sekundy = time(NULL);
      if(sekundy == -1) return ret;

      localtime_r(&sekundy, &czas);

      if((czas.tm_hour < 8) || (czas.tm_hour > 15)) {
         if(flags&PAM_SILENT) return odpowiedz;

         komunikat.msg_style = PAM_TEXT_INFO;
         komunikat.msg = tempptr =
                     malloc(strlen(info)+1);
         sprintf(tempptr,"%s",info);
         pam_get_item(pamh, PAM_CONV,
            (const void **)&rozmowa);
            rozmowa->conv(1, (const struct pam_message
            **)&pkomunikat, &resp,
            rozmowa->appdata_ptr);
         usleep(2000000);
         free(tempptr);

        if (resp)
            _pam_drop_reply(resp, 1);
      } else odpowiedz = PAM_SUCCESS;
   }
   return odpowiedz;
}
```

związanych z kontem logującego się użytkownika, na przykład jego imienia i nazwiska,

- *no_warn* – moduł nie powinien zgłaszać żadnych ostrzeżeń do wywołującego programu,
- *use_mapped_pass* – argument ten pozwala na użycie hasła podanego przez użytkownika w celu uzyskania przez moduł informacji o uwierzytelnianiu pochodzącej z innego źródła.

Jeśli funkcja przyjmie flagę `PAM_SILENT`, nie powinna zwracać do aplikacji żadnego tekstu (np. błędów lub informacji debugujących).

Współpraca z aplikacjami

Na razie mogło by się wydawać, że moduł PAM jest wielkim altruistą – wiele daje i nie bierze nic w zamian. Biblioteka *libpam* zapewnia zestaw funkcji niezbędnych do interakcji ze światem PAM – z innymi modułami oraz korzystającymi z nich programami. Ich prototypy i dokładny opis działania znajdziecie w dokumentacji, podczas gdy ja omówię jedynie ich znaczenie:

- `pam_set_data()`, `pam_get_data()` – służą do zapisu i odczytu stanu sesji; użycie zmiennych typu `static` nie jest wskazane, w zamian mamy do dyspozycji ten mechanizm,
- `pam_set_item()`, `pam_get_item()` – służą do zapisu i odczytu zmiennych `PAM_*`,
- `pam_get_user()` – wczytuje nazwę użytkownika,
- `pam_putenv()`, `pam_getenv()` – zarządzanie zmiennymi środowiskowymi PAM,
- `pam_getenvlist()` – wczytuje całą listę zmiennych środowiskowych związanych z PAM,
- `pam_strerror()` – formatuje komunikat o błędzie w oparciu o numer (kod) błędu,
- `pam_fail_delay()` – implementuje obsługę opóźnień po nieudanej autentykacji.

Przykładowe moduły

Poniżej omawiam kod dwóch prostych, małych modułów, które napisałem na potrzeby artykułu – aby pokazać działanie wspomnianych mechanizmów PAM. Źródła modułów znajdziecie również na płytce dołączonej do pisma. Zakładamy, że będą one ładowane dynamicznie, więc nie będę omawiał aspektów związanych z kompilacją statyczną (jest to dobrze opisane w dokumentacji). Kilka ważniejszych struktur z API PAM znajdziecie na Listingu 3.

Przykładowy moduł #1 – pam_czas

Pierwszy moduł (*pam_czas.c*, na Listingu 1) będzie służył do wpuszczania użytkowników innych niż root tylko w określonych godzinach (8-15) i będzie przeznaczony do pracy w grupie zarządzania kontem. Moglibyśmy również napisać go tak, żeby pracował w grupie *session* albo uwierzytelniającej – byłoby to również w pełni poprawne rozwiązanie.

Dozwolone godziny umieścimy na stałe w module, chociaż naturalnie można zrobić odczyt z pliku konfiguracyjnego, jeśli ktoś ma taką potrzebę. Nie ma większych przeciwwskazań, jeśli chodzi o korzystanie z różnych bibliotek (np. *math*) w modułach PAM – należy tylko pamiętać o prawidłowej kompilacji kodu źródłowego. Kompilator nie zawsze zgłosi wszystkie problemy – jeśli korzystamy z makr PAM, trzeba pamiętać o dołączeniu nagłówka *security/_pam_macros.h,* bo gcc nam o tym nie przypomni.

Listing 2. *pam_custom_motd.c*

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define PAM_SM_SESSION

#include <security/pam_modules.h>
#include <security/_pam_macros.h>

#define DEFAULT_MOTD "/etc/motd"
#define CUSTOM_MOTD_DIR "/etc/cmotd/"

PAM_EXTERN int pam_sm_open_session(pam_handle_t *pamh,
   int flags,
   int argc,
   const char **argv)
{
const char *uzytkownik = NULL;
char *tmpptr = NULL;
char *dfmotd = NULL, *cmpath = NULL, *plik = NULL;
struct pam_conv *rozmowa;
struct pam_message komunikat,
struct pam_message *pkomunikat = &komunikat;
struct pam_response *resp = NULL;
int fd, ret; struct stat st;
char bufor[1024];

   if(flags&PAM_SILENT) return PAM_IGNORE;

   ret = pam_get_user(pamh, &uzytkownik, NULL);
   if(ret != PAM_SUCCESS) return ret;

   for(;argc-- > 0; ++argv)
   {
      if(!strncmp(*argv,"default_motd=",13))
         dfmotd = (char *) strdup(13+*argv);
      if(!strncmp(*argv,"cmotd_path=",10))
         cmpath = (char *) strdup(10+*argv);
   }
   if(dfmotd == NULL) dfmotd = DEFAULT_MOTD;
   if(cmpath == NULL) cmpath = CUSTOM_MOTD_DIR;

   if(cmpath[strlen(cmpath)-1] == '/')
      snprintf(bufor,1023,"%s%s",cmpath,uzytkownik);
   else
      snprintf(bufor,1023,"%s/%s",cmpath,uzytkownik);

   if(stat(bufor,&st))
      snprintf(bufor,1023,"%s",dfmotd);
   if ((fd = open(bufor, O_RDONLY, 0)) >= 0)
   {
      if ((fstat(fd, &st) < 0) || !st.st_size)
      return PAM_IGNORE;
      komunikat.msg = tmpptr = malloc(st.st_size+1);
      if(!komunikat.msg) return PAM_IGNORE;
      read(fd, tmpptr, st.st_size);
      tmpptr[st.st_size] = '\0';
      close(fd);

  komunikat.msg_style = PAM_TEXT_INFO;
      pam_get_item(pamh, PAM_CONV,
         (const void **)&rozmowa);
      rozmowa->conv(1, (const struct
         pam_message **)&pkomunikat,
         &resp, rozmowa->appdata_ptr);
      free(tmpptr);

      if (resp)
         _pam_drop_reply(resp, 1);
   }
return PAM_SUCCESS;
}

PAM_EXTERN int pam_sm_close_session(
   pam_handle_t *pamh,
   int flags,
   int argc,
   const char **argv)
{
   return PAM_IGNORE;
}
```

Po zdefiniowaniu zmiennych ładujemy wskaźnik na nazwę użytkownika przyjętą przez PAM (funkcja `pam_get_user()`). Jeśli użytkownikiem jest *root*, zezwalamy mu na wstęp niezależnie od godziny. W przeciwnym wypadku sprawdzamy lokalny czas i porównujemy go z dozwolonymi godzinami logowania. Jeśli wolno teraz otworzyć sesję, zwracamy wartość `PAM_SUCCESS`. Jeśli nie, wyświetlamy przy użyciu mechanizmu *conversation* komunikat o błędzie.

Mechanizm *conversation* jest zalecanym sposobem komunikacji z użytkownikiem. Na potrzeby testowania modułu można posługiwać się np. `printf()`, lecz użycie takich funkcji w gotowym produkcie nie jest zalecane.

Funkcja `usleep()` pozwala upewnić się, że użytkownik zdąży przeczytać komunikat (mógłby on zostać zmazany przez następny moduł).

Należy pamiętać, że omawiany moduł pełni funkcję jedynie strażnika wejścia do systemu. PAM nie zajmuje się sesjami w toku, chyba że ktoś w ich trakcie potrzebuje autentykacji, autoryzacji, nowej sesji lub zmiany hasła.

Kompilację przeprowadzamy komendą:

```
gcc -shared modul.c -o modul.so.
```

Gotową bibliotekę należy skopiować do katalogu */lib/security/*. Aby przetestować moduł, możemy dopisać linijkę do pliku */etc/pam.d/system-auth* (RedHat), w grupie account:

```
account     required     /lib/security/pam_czas.so
```

Listing 3. *Wybrane struktury dostępne w API PAM:*

```
// Podstawowa struktura mechanizmu conversation:
struct pam_conv {
   int (*conv)
   (int num_msg,
   const struct pam_message **msg,
   struct pam_response **resp,
   void *appdata_ptr);
   void *appdata_ptr;
};

// Struktura pam_message:
struct pam_message {
   int msg_style;
   const char *msg;
};

// Struktura pam_response:
struct pam_response {
   char *resp;
   int resp_retcode;
};
```

Przykładowy moduł #2 – pam_custom_motd

Drugi moduł (*pam_custom_motd.c*) jest nieco bardziej skomplikowany niż wcześniejszy. Kod źródłowy jest oparty o moduł *pam_motd*, standardowo dostępny z PAM, jednak jest on wzbogacony o drobny, całkiem funkcjonalny szczegół. Jeśli zostanie znaleziony plik */etc/cmotd/USER* (gdzie USER jest loginem użytkownika otwierającego sesję), jest on wyświetlany jako *message of the day* (MOTD). W przeciwnym razie użytkownik ujrzy zawartość */etc/motd*.

Wykorzystujemy funkcję `pam_sm_open_session()`, bo wyświetlanie komunikatów przy logowaniu powinno odbywać się na etapie *session management*. W deklaracjach zmiennych umieszczamy wszystkie stosowne wpisy, w tym deklaracje dla struktury *conversation* (jak we wcześniejszym module) i dla tymczasowego kilobajtowego bufora, w którym będziemy przechowywać ścieżkę do katalogu (kilobajt powinien wystarczyć). Jeśli moduł nie jest zobowiązany siedzieć cicho (w tym wypadku idea traci sens i trzeba wyjść z funkcji), wczytujemy nazwę użytkownika, a następnie argumenty do modułu, podawane w odpowiednich plikach konfiguracyjnych PAM.

Z argumentów wczytujemy do zmiennych ścieżki domyślnego *motd* (parametr `default_motd=...`) i katalogu zawierającego *motd* dla poszczególnych użytkowników (parametr `cmotd_path=...`). Jeśli parametry nie zostały podane, będziemy używać domyślnych nazw zdefiniowanych na początku kodu. Tworząc pełną nazwę pliku do wyświetlenia sprawdzamy, czy użytkownik podał końcowy ukośnik w nazwie katalogu. Funkcja `stat` sprawdza, czy istnieje specjalny plik *motd* dla użytkownika – jeśli nie, będzie wyświetlany domyślny *motd*.

Następnie wczytujemy zawartość pliku do bufora, na który przy okazji wskazuje element `msg` struktury `pam_message`. Na koniec ustawiamy typ informacji na tekstową i wysyłamy ją przy użyciu funkcji `pam_conv->conv()`.

Kompilacja i instalacja przebiega prawie tak samo, jak przy module *pam_czas*. Zmienia się tylko linia, którą należy dopisać do pliku i jej położenie (na grupę *session*):

```
session     required     /lib/security/pam_custom_motd.so
```

Dobre praktyki programistyczne

Moduł powinien zachowywać się poprawnie w problematycznych sytuacjach, takich jak brak dostępu do pliku, brak pamięci itp. Zaleca się nadpisywanie losowymi danymi albo zerami wszelkich zmiennych, w których przechowywane były hasła. Uparty programista mógłby odzyskać pamięć zwolnioną po module i odczytać z niej poufne dane.

Autorzy dokumentacji PAM kierują naszą uwagę na różnicę między numerami ID zwracanymi przez funkcje `getuid()`, `geteuid()` oraz `pam_get_user()` – dlatego nie należy ich wszystkich wrzucać do jednego koszyka. Podają oni bardzo dobry przykład – użytkownik A używa programu *setuid* na użytkownika B, aby stać się użytkownikiem C. Funkcja `getuid` zwróci ID użytkownika A, `geteuid` użytkownika B, a `pam_get_user` – C.

Naturalnie istnieje szereg wskazówek nie związanych z bezpieczeństwem. Przeważnie moduły PAM są ładowane dynamicznie – nie należy zatem używać zmiennych typu `static`. Dobrą praktyką jest wrzucanie wszelkich informacji o błędach do logów systemowych. Użytkownik powinien być niepokojony jedynie niezbędnymi komunikatami, np. *Logowanie zabronione* albo *Logowanie nie powiodło się*.

W dokumentacji do PAM można znaleźć zalecane poziomy logowania dla poszczególnych grup komunikatów. W dokumentacji przypomina się również o konieczności inicjalizacji struktur używanych w funkcjach konwersacyjnych (*conversation*). Trzeba przewidzieć przypadek, w którym taka funkcja zwróci niepoprawne dane, albo w ogóle nic nie zwróci – dobrze zainicjalizowana struktura ułatwia jego identyfikację.

Końcowe wskazówki

Spośród dostępnych modułów PAM warto na początek przeanalizować źródła czterech z nich: *pam_permit*, *pam_deny*, *pam_issue* i *pam_warn*. Ich funkcje to odpowiednio: wpuszczanie każdego, nie wpuszczanie nikogo, wyświetlanie */etc/issue*, logowanie zmiennych PAM do pliku. Wspomniane moduły czytelnik znajdzie w źródłach PAM. Warto także wnikliwie przeczytać dokumentację, która jest może miejscami niejasno napisana (tylko miejscami), ale za to jest dosyć bogata. I – oczywiście – eksperymentować samemu.■

W Sieci:

- http://www.kernel.org/pub/linux/libs/pam/