KRZYSZTOF
RYCHLICKI – KICIOR

Narodziny procesu

Stopień trudności

Pisząc aplikację – bez względu na system – często napotykamy na potrzebę implementacji utworzenia przez nią nowego procesu. W artykule wyjaśniamy jak można to zrobić bez uszczerbku dla bezpieczeństwa aplikacji. Oprócz tego wykorzystamy tworzenie procesów do wybiórczego przydzielania użytkownikom uprawnień dla programów.

Pojęciem, które prędzej czy później musi pojawić się w słowniku informatyka – niezależnie od tego, czy jest programistą, administratorem, czy nawet hardware'owcem – jest *proces*. Nie zagłębiając się w fachowe definicje, można swobodnie powiedzieć, że proces w większości systemów jest utożsamiany z pojedynczą aplikacją *standalone*, czyli typowym programem uruchamianym na jednym komputerze. Nie należy mylić tego z aplikacjami rozproszonymi, które do działania zwykle wykorzystują wiele procesów, działających do tego najczęściej na wielu komputerach.

Na początku programistycznej przygody (choć dotyczy to też administratorów przygotowujących własne skrypty lub proste aplikacje) z reguły nie trzeba zapoznawać się z dość niszowym zagadnieniem, jakim jest tworzenie procesów przez aplikacje. Jest to problem o tyle ciekawy, że zależy od konkretnego systemu. Jak to zwykle bywa, inne podejście do tematu znajdziemy w systemie Windows, a inne w rodzinie Unix/Linux.

Windows – prostszy, ale czy funkcjonalny?

Najpierw, dla formalności, zajmiemy się sposobem tworzenia procesów w systemie Windows, gdyż nie oferuje on tyle możliwości, co systemy uniksowe. Nie wymaga on też uwzględniania specjalnych zasad bezpieczeństwa. Przykład zrealizujemy na przykładzie języka C# i platformy .NET, jako najbardziej natywnej i nowoczesnej zarazem platformy programistycznej dla Windows. Do obsługi procesów wykorzystuje się klasę `Process` z przestrzeni nazw `System.Diagnostics`. Po utworzeniu jej instancji należy określić kilka kluczowych właściwości i wywołać metodę `Start()` (przedstawia to Listing 1):

Oczywiście istnieją stosowne właściwości określające tryb uruchomienia aplikacji – np. `WindowStyle`, która pozwala na uruchomienie aplikacji w trybie ukrytym (widoczna jest wtedy tylko w Menedżerze Zadań – należy z rozwagą używać tej opcji!). Za pomocą tej klasy można otwierać też dokumenty – np. pliki tekstowe czy graficzne – za pomocą programów skojarzonych ze stosownymi rozszerzeniami. Wtedy można również skorzystać z właściwości `Verb`, określającej czynność do wykonania na pliku (np. *open* lub *print*; spis czynności możliwych do wykonania w przypadku danego procesu oferuje właściwość `Verbs`).

Reasumując, podstawowe problemy związane z bezpieczeństwem przy uruchamianiu nowych procesów w systemie Windows dotyczą:

- konieczności sprawdzania danych pobranych od użytkownika pod kątem poprawności (oczywiście, jeśli aplikacja daje użytkownikowi taką możliwość),
- wybierania odpowiednich trybów uruchomienia nowego procesu (tryb okna),
- obsługi wyjątków, powstających przy błędach braku uprawnień.

Warto pamiętać, że nowo tworzony proces nie zastępuje już istniejącego – jest to ważne

Z ARTYKUŁU DOWIESZ SIĘ

O kwestiach związanych z tworzeniem nowych procesów w systemach z rodziny Windows oraz Unix.

Jak tworzyć nowe procesy, dając możliwości i uprawnienia tylko w takim zakresie, jaki jest zamierzony i absolutnie niezbędny.

CO POWINIENEŚ WIEDZIEĆ

Znać zasady tworzenia procesów

Jak działa system operacyjny.

Mieć podstawową wiedzę z zakresu programowania, zwłaszcza w języku C++.

w kontekście możliwości systemów uniksowych, o czym poniżej.

Unix – pełna paleta możliwości

W systemach uniksowych uruchamiać nowe procesy można różnorodnie; zajmiemy się jednym z najbardziej złożonych sposobów, który zarazem pozwala na najwięcej – pisaniem bezpośrednio w języku C++, z użyciem standardowego kompilatora g++.

Dwie zasadnicze metody, jakie można wykorzystać w tej sytuacji, opierają się na dwóch różnych funkcjach (z poszczególnymi ich wariantami) – `fork()` i `exec()`. Różnica między nimi polega na ich wpływie na aplikację wywołującą – rodzica (`parent`). Funkcja `fork()` tworzy kopię procesu-rodzica, kopiując także całą pamięć wykorzystywaną przez rodzica. Jedyną istotną różnicą jest PID (*Process ID*), czyli identyfikator procesu – proces-dziecko (`child`) otrzymuje nowy PID, dzięki czemu można go rozróżnić z rodzicem. Odwrotnie jest w przypadku funkcji `exec()`. Tworzony proces zastępuje proces wywołujący, zaś identyfikator procesu pozostaje ten sam. Można więc powiedzieć, że proces-dziecko *podszywa się* pod swojego rodzica.

Ze względu na fakt, że obydwie te funkcje mają swoje wady, najczęściej wykorzystuje się ich połączenie. Aby osiągnąć efekt znany z systemu Windows, czyli utworzenie nowego procesu z osobną przestrzenią adresową, należy najpierw wywołać funkcję `fork()`, aby zarezerwować nowe miejsce dla nowego procesu (pamiętając, że na początku to miejsce zostanie wypełnione kopią procesu-rodzica) z nowym PID, a następnie wywołać `exec()`, która zamieni nowo utworzoną kopię na nowy proces, pozostając jednak w miejscu procesu-dziecka i z jego PID. Mogłoby się wydawać, że takie postępowanie nie ma sensu. Po co rozdzielać proces tworzenia na dwie części? Podstawową odpowiedzią jest istota komunikacji między procesami. Po wykonaniu powyżej opisanych operacji dla dwóch procesów-dzieci, dysponując ich PID, możemy umożliwić im komunikację między sobą, co jest istotą mechanizmu potoków, znanego i wykorzystywanego w systemach uniksowych. Innym ciekawym zastosowaniem funkcji `exec()` jest tworzenie programów typu *wrapper* – małych aplikacji, sprawdzających, czy użytkownik je wywołujący posiada pewne uprawnienia, a następnie wywołujących właściwą aplikację z odpowiednimi uprawnieniami.

Terminologia

- *Proces* – jest to instancja programu, wykonywana w trakcie pracy systemu operacyjnego.
- *Rodzic, dziecko* – gdy jeden proces tworzy inny proces, proces tworzący jest rodzicem, a proces tworzony – dzieckiem.
- `errno` – zmienna w języku C++, dostępna pod warunkiem dołączenia przed kompilacją do kodu źródłowego pliku nagłówkowego *errno.h*, która udostępnia kod błędu ostatnio wykonanej operacji (o ile błąd nastąpił).
- `suid` – bit, który może być ustawiany dla plików wykonywalnych i skryptów, określający, czy program/skrypt ma być wywoływany z uprawnieniami właściciela (`suid` ustawiony), czy aktywnego użytkownika (brak `suid`).
- `sgid` – odpowiednik bitu `suid` dla grup – jeżeli jest ustawiony, uruchamia program z uprawnieniami grupy posiadającej plik.

```
$ ./opakowany
1035 1035
$ ./wrapper opakowany
1035 1035
$ ./wrapper opakowany
1035 1010
$ ./wrapper opakowany
1010 1010
$
```

Rysunek 1. *Wszystkie cztery sytuacje opisane przy różnych konfiguracjach wrappera*

Listing 1. *Tworzenie nowego procesu w systemie Windows*

```
Process p = new Process();
p.StartInfo.FileName = @"c:\program.exe";
p.StartInfo.Arguments = "arguments";
p.Start();
```

Listing 2. *Najprostszy przykład wrappera wywołującego program (wrapper.cpp)*

```
#include <unistd.h>
#include <iostream>
#include <errno.h>

using namespace std;

int main(int argc, char *argv[], char *env[])
{
        if (argc<2)
                return 1;
        execve(argv[1],argv,env);
        int nr = errno;
        if (nr != 0)
                cout << "Error occured! "<<strerror(nr) << endl;
        return 1;
}
```

Listing 3. *Program opakowywany (opakowany.cpp)*

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

using namespace std;

int main()
{
 cout << getuid() << " " << geteuid() << endl;
 return 0;
}
```

Listing 4. *Fragment Listingu 2 po zmianie*

```
if (argc<2)
        return 1;
setreuid(geteuid(),geteuid());
execve(argv[1],argv,env);
int nr = errno;
```

Opakowanie – temat nie tylko dla ekologów

Wrapper, czyli program opakowujący, tworzy się dla dwóch głównych zastosowań. Po pierwsze, wrapper może sprawdzić specyficzne warunki, w jakich program ma być uruchamiany, np. nazwę użytkownika wywołującego program, argumenty, etc. Drugi powód to wywołanie programu z innymi uprawnieniami niż te posiadane przez użytkownika uruchamiającego program. Użytkownik taki jest nazywany rzeczywistym. Używa się też pojęcia efektywny, w stosunku do użytkownika, który jest właścicielem uruchamianego pliku. Jedynym dodatkowym wymaganiem w stosunku do uruchamianego programu jest posiadanie przez niego bitu `suid`. Bit ten można ustawić za pomocą polecenia `chmod +s nazwaProgramu`. Dzięki niemu program uruchamiany jest z uprawnieniami właściciela. Jeśli więc administrator ustawi na posiadanym przez siebie programie bit `suid`, musi liczyć się z potencjalnymi problemami związanymi z bezpieczeństwem. Z tego względu stosuje się nieco inne rozwiązanie. Zamiast dawać uprawnienia konkretnemu programowi, bit `suid` otrzymuje wrapper. Może on uruchomić wybraną aplikację z uprawnieniami efektywnego użytkownika, ale tylko po spełnieniu określonych reguł. Na przykład, można we wrapperze podać listę loginów użytkowników, którzy mogą uruchomić opakowywany program – lub też postawić jakieś inne warunki. Jest to niewątpliwie dużo bardziej roztropne niż ustawianie bitu `suid` bezpośrednio na wybranym programie. Listing 2. prezentuje przykładowy, najprostszy wrapper, który wywołuje program podany jako argument.

Należy pamiętać, aby po kompilacji programu ustawić dla niego bit `suid`. Jeśli została podana ścieżka do pliku, należy wywołać program spod tej lokalizacji za pomocą jednego z wariantów funkcji rodziny `exec()`, czyli `execve()`. Litery `v` i `e` oznaczają konieczność podania argumentów dla wywoływanej aplikacji (które stanowią po prostu kolejne argumenty, począwszy od trzeciego), zaś ostatnia tablica jest tablicą zmiennych środowiskowych – choć z reguły w nagłówku funkcji `main()` nie podaje się jej jawnie. Funkcje z rodziny `exec()` zwracają tylko -1 w przypadku niepowodzenia i umieszczają kod błędu w zmiennej `errno`. Proszę jednak zwrócić uwagę, że w przypadku prawidłowego wywołania programu kod umieszczony za wywołaniem `execve()` nie zostanie wykonany!

Program opakowywany (Listing 2), również w języku C++, jest jeszcze krótszy.

Listing 5. *Wrapper blokujący niedozwolonych użytkowników*

```
#include <unistd.h>
#include <iostream>
#include <errno.h>
#include <sys/types.h>
#include <pwd.h>

using namespace std;

int main(int argc, char *argv[], char *env[])
{
        if (argc<2)
                return 1;
        uid_t id = getuid();
        passwd *p = getpwuid(id);
        if (strcmp(p->pw_name,"alloweduser")==0)
        {
                setreuid(geteuid(),geteuid());
                execve(argv[1],argv,env);
                int nr = errno;
                if (nr != 0)
                      cout << "Error occured! "<<strerror(nr) << endl;
        } else
                cout << "Access denied!" << endl;
        return 1;
}
```

Listing 6. *Sprawdzanie przynależności użytkownika do grupy*

```
#include <unistd.h>
#include <iostream>
#include <errno.h>
#include <sys/types.h>
#include <pwd.h>

using namespace std;

int main(int argc, char *argv[], char *env[])
{
        if (argc<2)
                return 1;
        uid_t id = getuid();
        passwd *p = getpwuid(id);
        int gID = p->pw_gid;
        group *grp = getgrgid(gID);
        if (strcmp(grp->gr_name,"allowedgroup")==0)
        {
                setreuid(geteuid(),geteuid());
                execve(argv[1],argv,env);
                int nr = errno;
                if (nr != 0)
                      cout << "Error occured! "<<strerror(nr) << endl;
        } else
                cout << "Access denied!" << endl;
        return 1;
}
```

W Sieci

- *http://msdn.microsoft.com/library/en-us/system.diagnostics.process.aspx* – dokumentacja klasy Process odpowiedzialnej za obsługę procesów w systemie Windows (.NET) – w jęz. angielskim,
- *http://linux.die.net/man/2/execve* – opis funkcji `execve()` oraz zasad działania bitu `suid`,
- *http://linux.die.net/man/2/fork* – opis funkcji `fork()` z wyszczególnionymi zasadami kopiowania procesu.

Program wyświetla dwa różne identyfikatory użytkownika – rzeczywisty i efektywny. Załóżmy, że ID użytkownika uruchamiającego program (czyli rzeczywistego) to 1035, a ID właściciela – 1010. Jeśli wywołamy ten program poleceniem:

```
./opakowany
```

to otrzymamy dwie identyczne liczby – 1035. Jeśli wywołamy ten program przed ustawieniem bitu `suid` na wrapperze:

```
./wrapper opakowany
```

otrzymamy taki sam rezultat. Jeśli jednak przed uruchomieniem wrappera bit `suid` zostanie ustawiony, wtedy otrzymamy na wyjściu wynik: `1035 1010`

Listing 7. *Ostateczny sposób weryfikacji użytkownika uruchamiającego program*

```
#include <unistd.h>
#include <iostream>
#include <errno.h>
#include <sys/types.h>
#include <pwd.h>

using namespace std;

int main(int argc, char *argv[], char
                        *env[])
{
        if (argc<2)
                return 1;
        uid_t id = getuid();
        passwd *p = getpwuid(id);
        group *grp = getgrnam("allowedg
                        roup");
        int i = 0;
        while (grp->gr_mem[i])
        {
             if (strcmp(grp->gr_mem[i],p-
                        >pw_name)==0)
             {
                    setreuid(geteuid(),
                      geteuid());
                    execve(argv[1],argv
                      ,env);
                    int nr = errno;
                    if (nr != 0)
                          cout <<
                      "Error occured!
                      "<<strerror(nr)
                      << endl;
                    break;
             }
             i++;
        }
        return 1;
}
```

Jak widać, została dokonana zmiana efektywnego użytkownika, czego nie można powiedzieć o rzeczywistym. Czy da się zatem zrobić coś, aby program widział, że użytkownik rzeczywisty to de facto użytkownik efektywny?

Oczywiście. Wystarczy wywołać funkcję `setreuid()`, która ustawia kolejno ID użytkownika rzeczywistego i efektywnego według podanych kolejno argumentów funkcji. Wystarczy więc zmodyfikować kod, jak na Listingu 4.

Dzięki dodaniu wywołania funkcji `setreuid()` uzyskujemy pożądany efekt. Wywołanie wrappera na programie opakowywanym na koncie naszego testowego użytkownika o ID równym 1035 spowoduje wyświetlenie następującego rezultatu: `1010 1010`

Wszystkie możliwości zostały przedstawione na Rysunku 1.

To możliwości, a gdzie bezpieczeństwo?

Powyżej opisane zostało bardzo ważne ułatwienie, pozwalające na przekazywanie dostępu i uprawnień do niektórych aplikacji. Jednak nasz kod nie wprowadza rozgraniczenia, kto może wywołać funkcję `execve()`, a kto nie – a zatem, kto zyskuje dostęp do aplikacji. Wiemy, jak można pobrać identyfikatory aktualnego użytkownika; ostatnim krokiem pozostało wydobycie pozostałych informacji na jego temat, wykorzystując właśnie jego ID.

Do tego celu można wykorzystać m.in. funkcję `getpwuid()`, która zwraca wskaźnik do struktury `passwd`. Dzięki niej na podstawie ID użytkownika możemy poznać jego login oraz identyfikator grupy, do której należy. Takie informacje dają podstawy do stworzenia odpowiednich warunków, ograniczających dostęp do uruchamiania programów opakowanych. Proste zabezpieczenie prezentuje Listing 5.

Wykorzystujemy w nim wspomnianą wcześniej funkcję, aby uzyskać dostęp do struktury użytkownika. Zwyczajne porównanie loginów pozwala orzec o dopuszczeniu do wywołania programu z uprawnieniami lub jego braku. Zauważmy, że istnieje szereg bardziej wyrafinowanych możliwości, jak choćby sprawdzanie przynależności do grup – wszystko to można obsłużyć przy pomocy funkcji z rodziny `getpwuid()`, takich jak np. `getgrgid()` czy `getgrnam()`. Można zweryfikować, czy użytkownik należy do danej grupy – nie będzie trzeba wtedy wyliczać wszystkich dozwolonych użytkowników, a tylko jedną grupę, do której będą należeć użytkownicy dopuszczeni do uruchamiania programu. Prosty przykład takiego rozwiązania prezentuje Listing 6.

Aby pobrać nazwę grupy, do której należy użytkownik, należy znać jej ID. Łącząc go z funkcją `getgrgid()`, uzyskujemy pożądaną nazwę. Jednak powyższe rozwiązanie jest poprawne tylko w części. Otóż ID grupy, które znajduje się we właściwości `pw_gid`, odnosi się do głównej grupy, do której należy użytkownik. Nic zaś nie stoi na przeszkodzie, aby użytkownik należał do wielu grup. Jak rozwikłać ten, ostatni już, dręczący nas problem?

Dysponując egzemplarzem struktury `grp`, możemy skorzystać z jej właściwości `char **gr_mem`. Jest to lista łańcuchów będących loginami użytkowników należących do danej grupy. Wystarczy więc przejrzeć tę listę w poszukiwaniu loginu użytkownika wywołującego wrapper, co prezentuje Listing 7.

Tym razem do pobrania informacji o żądanej grupie musimy wykorzystać inną funkcję z tej samej rodziny – chyba, że znamy ID grupy. Następnie iterujemy po kolejnych loginach użytkowników należących do grupy i sprawdzamy, czy nie trafiliśmy przypadkiem na poszukiwanego użytkownika. Przerwanie pętli w przypadku znalezienia użytkownika jest konieczne, aby nie wykonywać niepotrzebnych iteracji.

Podsumowanie

Artykuł z pewnością nie wyczerpuje tematu; zaprezentowane przykłady pokazują ledwie zarys możliwości, jakie może uzyskać programista/administrator, który zechce bezpiecznie uruchamiać nowe procesy, nie chcąc być zmuszonym do całkowitego zablokowania tej możliwości dla użytkowników systemu. Warto zwrócić uwagę na strony opisane w ramce W Sieci, aby poznać wszystkie niuanse tworzenia nowych procesów.

Krzysztof Rychlicki – Kicior
Krzysztof programuje w Javie, C# i Pythonie. Jest autorem książek, m.in. C#. Tworzenie aplikacji graficznych w .NET 3.0 (Wydawnictwo Helion, 2007) oraz wielu artykułów z zakresu Delphi, PHP, J2ME, C# i Pythona.
Kontakt z autorem: *kitikatpl@gmail.com*