21A

Zabezpieczenia dostępu kodu

Code Access Security

Technologia zabezpieczania dostępu kodu (ang. *Code Access Security* — CAS) umożliwia środowisku CLR utworzenie ograniczonego środowiska wykonawczego, w którym nie ma możliwości wykonywania pewnych rodzajów operacji (na przykład odczytywania plików systemu operacyjnego, z repertuaru refleksji czy tworzenia interfejsu użytkownika). Utworzone przez CAS ograniczone środowisko wykonawcze nazywa się **środowiskiem wykonawczym o ograniczonym zaufaniu**, podczas gdy normalne nazywa się **środowiskiem wykonawczym o pełnym zaufaniu**.

Technologia CAS odgrywała strategiczną rolę w początkach platformy .NET, ponieważ umożliwiała:

- wykonywanie kontrolek ActiveX w języku C# w przeglądarce internetowej (jak apletów Javy);
- obniżenie kosztów ponoszonych na utrzymywanie współdzielonych hostingów przez danie możliwości uruchamiania wielu witryn w tym samym procesie .NET;
- wdrażanie aplikacji ClickOnce o ograniczonych uprawnieniach przez internet.

Dwa pierwsze punkty są już nieistotne, a trzeci zawsze był wątpliwy, ponieważ użytkownik końcowy najczęściej nie orientuje się w kwestiach dotyczących ograniczania uprawnień przed instalacją programu. I chociaż funkcja CAS wciąż ma pewne zastosowania, to najczęściej jest używana w różnych specjalnych przypadkach. Problematyczna jest też niezawodność ograniczonego środowiska tworzonego przez funkcję CAS. W 2015 r. firma Microsoft oświadczyła, że CAS nie można traktować jako mechanizmu wyznaczania granic systemu zabezpieczeń (poza tym CAS w dużej części usunięto z .NET Standard 2.0). Te wszystkie trudności istnieją mimo ulepszeń funkcji CAS wprowadzonych wraz z CLR 4 w 2010 r.

Natomiast technika tworzenia ograniczonych środowisk wykonawczych z pominięciem funkcji CAS wciąż jest z powodzeniem wykorzystywana. W środowisku takim działają aplikacje UWP, podobnie jak biblioteki CLR SQL. Tworzenie tych środowisk wymusza system operacyjny lub hostowane środowisko CLR. Są one bardziej niezawodne od środowisk CAS oraz łatwiejsze do

utworzenia i obsługi. Zabezpieczenia systemu operacyjnego współpracują także z kodem niezarządzanym, więc aplikacja UWP nie ma możliwości odczytywania ani zapisywania dowolnych plików, czy to w języku C#, czy C++.

Z tych powodów zrezygnowaliśmy z przedstawiania opisu technologii CAS w książce *C# 7.0 w pigułce*, ale materiał z poprzedniego wydania zamieściliśmy na serwerze FTP wydawnictwa Helion. (Twórcy bibliotek nadal mogą potrzebować obsługi częściowo zaufanych środowisk, jeśli zależy im na współpracy ze starymi platformami).

Opisane w tym rozdziale typy należą do następujących przestrzeni nazw:

```
System.Security;
System.Security.Permissions;
System.Security.Principal;
System.Security.Cryptography;
```

Uprawnienia

Uprawnienia opisaliśmy w rozdziale 21. książki *C# 7.0 w pigułce* w kontekście zabezpieczeń opartych na tożsamościach i rolach. Teraz wracamy do nich w kontekście technologii CAS.

W platformie .NET Framework uprawnienia są wykorzystywane zarówno do tworzenia ograniczonego środowiska wykonawczego, jak i do autoryzacji. **Uprawnienie** jest czymś w rodzaju bramy, która warunkowo uniemożliwia wykonanie kodu. W ograniczonym środowisku wykonawczym stosowane są uprawnienia **dostępu kodu**. Natomiast w technikach autoryzacji używa się uprawnień **tożsamości** i **ról**.

Choć obie technologie opierają się na podobnym modelu, różnią się znacznie pod względem sposobu użycia. Częściowo dzieje się tak dlatego, że korzystający z nich programista raz jest z jednej, a raz z innej strony barykady. W przypadku ograniczeń dostępu kodu programista jest tą stroną, która *nie cieszy się zaufaniem*. Natomiast w przypadku zabezpieczania tożsamości i ról jesteśmy tą stroną, która *nie darzy innych zaufaniem*. Bezpieczeństwo dostępu kodu najczęściej wymusza na nas CLR lub środowisko hostingowe, takie jak ASP.NET lub Internet Explorer. Natomiast autoryzację implementuje się w celu uniemożliwienia niepowołanym osobom dostępu do naszego programu.

CodeAccessPermission i PrincipalPermission

Wyróżnia się dwa zasadnicze rodzaje uprawnień:

`CodeAccessPermission`
: Jest to abstrakcyjna klasa bazowa dla wszystkich klas uprawnień CAS, takich jak `FileIOPer`
`↪mission`, `ReflectionPermission` czy `PrintingPermission`.

`PrincipalPermission`
: Opisuje tożsamość i/lub rolę (np. `"Maria"` lub `"Zasoby ludzkie"`).

W przypadku klasy `CodeAccessPermission` termin *uprawnienie* może być trochę mylący, ponieważ sugeruje że coś zostało komuś przyznane. Jednak wcale nie musi to być prawdą. Obiekt klasy `CodeAccessPermission` opisuje **uprzywilejowaną operację**.

Na przykład obiekt klasy `FileIOPermission` opisuje uprawnienie do tego, że coś można odczytywać, zapisywać oraz dołączać do wybranego zbioru plików lub katalogów. Obiektu takiego można użyć na wiele sposobów:

- w celu sprawdzenia, czy bieżący moduł i wszystkie nadrzędne moduły mają prawo do wykonania tych operacji (`Demand`);
- w celu sprawdzenia, czy bezpośredni wywołujący moduł nadrzędny ma prawo wykonywać te czynności (`LinkDemand`);
- w celu tymczasowego opuszczenia ograniczonego środowiska wykonawczego i utrzymania praw nadanych przez zestaw, aby wykonać czynności niezależnie od uprawnień nadrzędnego modułu wywołującego.

W CLR można też znaleźć następujące akcje zabezpieczeń: `Deny`, `RequestMinimum`, `RequestOptional`, `RequestRefuse` oraz `PermitOnly`. Jednak w .NET Framework 4.0 nadano im status wycofywanych na rzecz nowego modelu **transparentności**.

Klasa `PrincipalPermission` jest znacznie prostsza. Zawiera tylko jedną metodę zabezpieczeń o nazwie `Demand`, która sprawdza, czy określony użytkownik lub określona rola są prawidłowe dla bieżącego wątku wykonawczego.

Interfejs IPermission

Both `CodeAccessPermission` i `PrincipalPermission` implementują interfejs `IPermission`:

```
public interface IPermission
{
  void Demand();
  IPermission Intersect (IPermission target);
  IPermission Union (IPermission target);
  bool IsSubsetOf (IPermission target);
  IPermission Copy();
}
```

Kluczowe znaczenie ma tu metoda `Demand`. Sprawdza, czy dana operacja wymagająca podniesionych uprawnień rzeczywiście ma takie uprawnienia, i zgłasza wyjątek `SecurityException`, jeśli stwierdzi, że jej ich brakuje. Ten, kto jest stroną *nieufającą*, wywołuje metodę `Demand`. Natomiast strona *niezaufana* podlega kontroli za pomocą tej metody.

Aby np. pozwolić tylko Marii na tworzenie raportów, można napisać taki kod:

```
new PrincipalPermission ("Maria", null).Demand();
// ... sporządzanie raportu
```

Dla porównania powiedzmy, że nasz zestaw działa w ograniczonym środowisku, w którym zabronione jest wykonywanie plikowych operacji wejścia i wyjścia, przez co poniższy wiersz kodu powoduje wyjątek `SecurityException`:

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
...
```

W tym przypadku metoda Demand jest wywoływana przez kod, który my wywołujemy, czyli konstruktor klasy FileStream:

```
...
new FileIOPermission (...).Demand();
```

Metoda Demand analizuje cały stos wywołań, aby sprawdzić, czy odpowiednie uprawnienia mają wszystkie elementy łańcucha wywołań (w obrębie bieżącej domeny aplikacji). Zasadniczo można powiedzieć, że zadaje następujące pytanie: „Czy ta domena aplikacji może korzystać z tego uprawnienia?”.

Zabezpieczenia dostępu kodu powodują, że czasami powstają ciekawe przypadki z zestawami, które są uważane za *w pełni zaufane*. Jeżeli taki zestaw działa w ograniczonym środowisku, wszelkie wykonywane przez niego wywołania metody Demand podlegają pod zestaw uprawnień tego środowiska. Ale w pełni zaufane zestawy mogą tymczasowo *uciekać* z piaskownicy przez wywołanie metody Assert na obiekcie klasy CodeAccess ↪Permission. Wówczas wszystkie wywołania metody Demand dotyczące uprawnień, których się domagano, będą miały pozytywny skutek. Metoda Assert kończy działanie wraz z bieżącą metodą lub po wywołaniu przez programistę metody CodeAccess ↪Permission.RevertAssert.

Metody Intersect i Union łącza dwa obiekty o takich samych uprawnieniach w jeden. Pierwsza z nich tworzy „mniejszy” obiekt uprawnień, podczas gdy druga tworzy „większy” obiekt uprawnień.

Jeśli chodzi o uprawnienia dostępu kodu, „większy” obiekt uprawnień jest *bardziej* restrykcyjny w stosunku do metody Demand, ponieważ wymaga posiadania większej liczby uprawnień.

(Jeśli chodzi o uprawnienia PrincipalPermission, „większy” obiekt uprawnień jest *mniej* restrykcyjny w stosunku do metody Demand, ponieważ wystarczy *jedna* z zasad lub tożsamości, aby spełnić żądanie).

Klasa PermissionSet

Klasa PermissionSet reprezentuje kolekcję różnych typów obiektów implementujących interfejs IPermission. Poniżej znajduje się przykład utworzenia takiego zbioru zawierającego trzy uprawnienia dostępu kodu i wywołania na nich wszystkich naraz metody Demand.

```
PermissionSet ps = new PermissionSet (PermissionState.None);

ps.AddPermission (new UIPermission (PermissionState.Unrestricted));
ps.AddPermission (new SecurityPermission (
                      SecurityPermissionFlag.UnmanagedCode));
ps.AddPermission (new FileIOPermission (
                      FileIOPermissionAccess.Read, @"c:\docs"));
ps.Demand();
```

Konstruktor klasy PermissionSet przyjmuje wyliczenie PermissionState, które wskazuje, czy dany zbiór ma podlegać jakimkolwiek ograniczeniom. Zbiór uprawnień bez ograniczeń jest traktowany tak, jakby zawierał wszystkie możliwe uprawnienia (choć jego kolekcja jest pusta). Zestawy wykonywane bez ograniczeń dostępu kodu traktowane są jako *w pełni zaufane*.

Metoda AddPermission posługuje się semantyką podobną do metody Union — też tworzy zbiór „większy”. Wywołanie tej metody na zbiorze uprawnień bez ograniczeń nie daje żadnego efektu (ponieważ logicznie zbiór ten ma już wszelkie możliwe uprawnienia).

Na zbiorach uprawnień można wywoływać metody `Union` i `Intersect`, tak jak na wszystkich obiektach implementujących interfejs `IPermission`.

Bezpieczeństwo deklaratywne i imperatywne

Do tej pory ręcznie tworzyliśmy obiekty uprawnień i wywoływaliśmy na nich metodę `Demand`. Jest to przykład **bezpieczeństwa imperatywnego**. Taki sam efekt można uzyskać przez dodanie atrybutów do metody, konstruktora, klasy, struktury lub zestawu — to jest bezpieczeństwo **deklaratywne**. Choć pierwszy rodzaj bezpieczeństwa jest elastyczniejszy, bezpieczeństwo deklaratywne ma trzy zalety:

- potencjalnie mniejsza ilość kodu;
- możliwość stwierdzenia przez CLR z góry, jakie uprawnienia są potrzebne zestawowi;
- możliwość optymalizacji wydajności.

Na przykład:

```
[PrincipalPermission (SecurityAction.Demand, Name="Mary")]
public ReportData GetReports()
{
  ...
}

[UIPermission(SecurityAction.Demand, Window=UIPermissionWindow.AllWindows)]
public Form FindForm()
{
  ...
}
```

Ta technika bazuje na fakcie, że każdy typ uprawnień ma w .NET Framework siostrzany typ atrybutu. Odpowiednikiem typu `PrinciplePermission` jest zatem atrybut `PrincipalPermissionAttribute`. Pierwszym argumentem konstruktora atrybutu zawsze jest obiekt typu `SecurityAction` wskazujący, którą metodę zabezpieczeń wywołać po utworzeniu obiektu uprawnień (najczęściej jest to metoda `Demand`). Pozostałe parametry odpowiadają własnościom odpowiednich obiektów uprawnień.

Zabezpieczenia dostępu kodu

W tabelach 21a.1 – 21a.6 przedstawiono typy `CodeAccessPermission` wykorzystywane w platformie .NET Framework. Z założenia wszystkie razem powinny się odnosić do wszystkich możliwych sposobów nieprawidłowego zachowywania się programów!

Tabela 21a.1. Uprawnienia podstawowe

Typ	Możliwości
`SecurityPermission`	Zaawansowane operacje, np. wywoływanie kodu niezarządzanego
`ReflectionPermission`	Korzystanie z mechanizmów refleksji
`EnvironmentPermission`	Odczytywanie i zapisywanie ustawień środowiska wiersza poleceń
`RegistryPermission`	Odczytywanie i zapisywanie rejestru systemu Windows

SecurityPermission przyjmuje argument SecurityPermissionFlag. Jest to wyliczenie umożliwiające zastosowanie dowolnej kombinacji poniższych ustawień:

```
AllFlags                ControlThread
Assertion               Execution
BindingRedirects        Infrastructure
ControlAppDomain        NoFlags
ControlDomainPolicy     RemotingConfiguration
ControlEvidence         SerializationFormatter
ControlPolicy           SkipVerification
ControlPrincipal        UnmanagedCode
```

Najważniejszą składową tego wyliczenia jest Execution, która decyduje o możliwości wykonywania kodu. Pozostałe z tych uprawnień należy nadawać tylko, gdy w pełni ufa się wykonywanemu programowi, ponieważ umożliwiają wyjście z ograniczonego środowiska wykonawczego. Opcja ControlApp
↪Domain umożliwia tworzenie nowych domen aplikacji (rozdział 24.), a UnmanagedCode pozwala na wywoływanie metod macierzystych (rozdział 25.).

ReflectionPermission przyjmuje wyliczenie RefelctionPermissionFlag, które zawiera składowe MemberAccess i RestrictedMemberAccess. Jeśli programista uruchamia zestawy w środowisku ograniczonym, to drugie z tych ustawień jest bezpieczniejsze, gdy trzeba umożliwić korzystanie z mechanizmów refleksji potrzebnych np. interfejsom API takim jak LINQ to SQL.

Tabela 21a.2. Uprawnienia do wykonywania operacji wejścia i wyjścia oraz operacji na danych

Typ	Możliwości
FileIOPermission	Odczytywanie i zapisywanie plików i katalogów
FileDialogPermission	Odczytywanie i zapisywanie plików wybranych za pomocą okna dialogowego *Otwieranie* lub *Zapisywanie*
IsolatedStorageFilePermission	Odczytywanie i zapisywanie danych we własnym izolowanym magazynie
ConfigurationPermission	Odczytywanie plików konfiguracyjnych aplikacji
SqlClientPermission, OleDbPermission, OdbcPermission	Komunikacja z serwerem baz danych przy użyciu klasy SqlClient, OleDb lub Odbc
DistributedTransactionPermission	Udział w transakcjach rozproszonych

FileDialogPermission kontroluje dostęp do klasy OpenFileDialog i SaveFileDialog. Klasy te są zdefiniowane w przestrzeniach nazw Microsoft.Win32 (przeznaczona do użytku przez aplikacje WPF) i System.Windows.Forms (do użytku w aplikacjach Windows Forms). Dodatkowo potrzebny jest obiekt UIPermission. Nie ma natomiast wymogu dodania obiektu FileIOPermission, jeśli ktoś ma dostęp do wybranego pliku przez wywołanie metody OpenFile na obiekcie OpenFileDialog lub SaveFileDialog.

Posiadanie tych uprawnień jest weryfikowane przez platformę .NET Framework. Istnieją też klasy uprawnień, w przypadku których weryfikacja uprawnień jest egzekwowana w kodzie programisty. Najważniejsze z nich dotyczą ustalania tożsamości zestawu wywołującego i zostały wymienione w tabeli 21a.7. Haczyk jest taki, że (jak w przypadku wszystkich uprawnień dostępu kodu) metoda Demand zawsze zwraca prawdę, jeżeli domena aplikacji ma pełne zaufanie (zob. następną sekcję).

Tabela 21a.3. Uprawnienia sieciowe

Typ	Możliwości
`DnsPermission`	Wyszukiwanie DNS
`WebPermission`	Dostęp do sieci dzięki obiektowi klasy `WebRequest`
`SocketPermission`	Dostęp do sieci dzięki obiektowi klasy `Socket`
`SmtpPermission`	Wysyłanie wiadomości e-mail przy użyciu bibliotek SMTP
`NetworkInformationPermission`	Możliwość korzystania z takich klas, jak `Ping` i `NetworkInterface`

Tabela 21a.4. Uprawnienia dotyczące szyfrowania

Typ	Możliwości
`DataProtectionPermission`	Możliwość posługiwania się metodami ochrony danych systemu Windows
`KeyContainerPermission`	Szyfrowanie przy użyciu klucza publicznego i składanie podpisów
`StorePermission`	Dostęp do magazynów certyfikatów X.509

Tabela 21a.5. Uprawnienia dotyczące interfejsu użytkownika

Typ	Możliwości
`UIPermission`	Tworzenie okien i interakcja ze schowkiem
`WebBrowserPermission`	Możliwość posługiwania się kontrolką `WebBrowser`
`MediaPermission`	Obsługa grafiki, audio i wideo w WPF
`PrintingPermission`	Dostęp do drukarki

Tabela 21a.6. Uprawnienia diagnostyczne

Typ	Możliwości
`EventLogPermission`	Odczytywanie lub zapisywanie dzienników zdarzeń systemu Windows
`PerformanceCounterPermission`	Możliwość korzystania z mierników wydajności systemu Windows

Tabela 21a.7. Uprawnienia dotyczące tożsamości

Typ	Narzuca
`GacIdentityPermission`	Zestaw jest ładowany do GAC
`StrongNameIdentityPermission`	Zestaw wywołujący ma określoną silną nazwę
`PublisherIdentityPermission`	Zestaw wywołujący ma podpis Authenticode na podstawie określonego certyfikatu

Stosowanie zasad dostępu kodu

Kiedy plik wykonywalny .NET jest uruchamiany z poziomu powłoki systemu Windows lub wiersza poleceń, ma nieograniczone uprawnienia. Sytuację taką nazywa się **pełnym zaufaniem**.

Jeżeli ktoś uruchomi zestaw poprzez inne środowisko — np. hosta integracji CLR serwera SQL Server, ASP.NET, ClickOnce lub niestandardowego hosta — host ten decyduje, jakie uprawnienia nadać zestawowi. Jeżeli ogranicza on je w jakikolwiek sposób, nazywa się to **zaufaniem ograniczonym** lub **uruchomieniem w środowisku ograniczonym**.

Mówiąc dokładniej, host nie ogranicza uprawnień naszego *zestawu*, tylko tworzy domenę aplikacji o ograniczonych uprawnieniach i do niej ładuje nasz zestaw. Oznacza to, że wszystkie inne zestawy ładowane do tej domeny (np. używane przez nasz zestaw) także działają w tym samym ograniczonym środowisku. Są jednak dwa wyjątki od tej reguły:

- zestawy zarejestrowane w GAC (wliczając w to .NET Framework);
- zestawy obdarzone przez hosta pełnym zaufaniem.

Należące do tych dwóch kategorii zestawy są uważane za *w pełni zaufane* i mogą wychodzić z ograniczonego środowiska przez zażądanie przez metodę `Assert` dowolnych uprawnień. Ponadto mogą wywoływać metody oznaczone jako `[SecurityCritical]` z innych w pełni zaufanych zestawów, uruchamiać nieweryfikowalny (niebezpieczny) kod oraz wywoływać metody egzekwujące żądania połączenia, które zawsze będą udane.

Kiedy więc mówimy, że *częściowo zaufany* zestaw wywołuje *w pełni zaufany* zestaw, mamy na myśli, że zestaw działający w ograniczonej domenie aplikacji wywołuje zestaw GAC — lub zestaw oznaczony przez hosta jako w pełni zaufany.

Sprawdzanie poziomu zaufania

Za pomocą poniższego kodu można sprawdzić, czy posiadane uprawnienia są nieograniczone:

```
new PermissionSet (PermissionState.Unrestricted).Demand();
```

Jeśli domena aplikacji jest ograniczona, powyższe wyrażenie spowoduje wyjątek. Ale może się też zdarzyć, że zestaw jednak będzie w pełni zaufany i będzie mógł wyjść z piaskownicy za pomocą metody `Assert`. Aby sprawdzić, czy istnieje taka możliwość, należy wysłać zapytanie do własności `IsFullyTrusted` interesującego zestawu.

Dopuszczanie częściowo zaufanych wywołujących

Możliwość akceptowania przez zestawy częściowo zaufanych wywołujących sprawia, że można przeprowadzić atak polegający na nielegalnym zdobyciu wysokich uprawnień, i jest w związku z tym zablokowana przez CLR, chyba że programista zażąda inaczej. Aby zrozumieć, dlaczego tak się dzieje, trzeba wiedzieć, na czym polegają ataki polegające na podnoszeniu uprawnień.

Podnoszenie uprawnień

Powiedzmy, że system CLR nie egzekwuje przestrzegania opisanej wcześniej reguły i napisaliśmy bibliotekę przeznaczoną do wykorzystania w programach o pełnym zaufaniu. Jedna z własności wygląda tak:

```
public string ConnectionString
  => File.ReadAllText (_basePath + "cxString.txt");
```

Teraz załóżmy, że użytkownik, który wdraża naszą bibliotekę, postanawia (słusznie bądź nie) załadować nasz zestaw do GAC. Następnie użytkownik ten uruchamia kompletnie niepowiązaną z naszą biblioteką aplikację hostowaną w ClickOnce lub ASP.NET w ograniczonym środowisku uruchomieniowym. Ta aplikacja ładuje nasz w pełni zaufany zestaw i próbuje wywołać własność `Connection ↪String`. Na szczęście kończy się to wyjątkiem `SecurityException`, ponieważ metoda `File.Read ↪AllText` będzie żądać uprawnienia `FileIOPermission`, którego wywołujący nie ma (przypomnijmy, że metoda `Demand` sprawdza stos wywołań). A teraz spójrz na poniższą metodę:

```
public unsafe void Poke (int offset, int data)
{
  int* target = (int*) _origin + offset;
  *target = data;
  ...
}
```

Bez niejawnego wywołania metody `Demand` zestaw działający w ograniczonym środowisku może wywołać tę metodę — i wykorzystać ją do złych celów. Na tym właśnie polega atak **podnoszenia uprawnień**.

Problem w tym przypadku polega na tym, że programista biblioteki nigdy nie planował, że będzie ona używana przez częściowo zaufane zestawy. Na szczęście system CLR pomaga nam w takiej sytuacji.

Atrybuty APTCA i [SecurityTransparent]

Aby pomóc w udaremnianiu ataków polegających na podnoszeniu uprawnień, system CLR nie pozwala częściowo zaufanym zestawom na domyślne wywoływanie w pełni zaufanych zestawów[1].

Aby móc wykonywać takie wywołania, należy zrobić jedną z dwóch rzeczy z w pełni zaufanym zestawem:

- zastosować atrybut `[AllowPartiallyTrustedCallers]` (w skrócie zwany APTCA);
- zastosować atrybut `[SecurityTransparent]`.

Stosując te atrybuty, należy się liczyć z możliwością bycia stroną *nieufającą* (a nie: *niezaufaną*).

Przed CLR 4.0 obsługiwany był tylko atrybut APTCA i jego jedyną funkcją było umożliwianie działania częściowo zaufanych wywołujących. Od CLR 4.0 atrybut ten dodatkowo niejawnie oznacza wszystkie metody (i funkcje) w zestawie jako **transparentne pod względem zabezpieczeń**. Szczegółowo wyjaśniamy to w następnej sekcji, a na razie wystarczy wiedzieć, że metody nie mogą wykonywać żadnej z poniższych czynności (zarówno przy pełnym, jak i częściowym zaufaniu):

- uruchamiać nieweryfikowalnego (niebezpiecznego) kodu;
- uruchamiać kodu macierzystego poprzez P/Invoke ani COM;
- żądać uprawnień, aby podnieść swój poziom zabezpieczeń;
- zaspokajać żądania konsolidacji;

[1] W wersjach CLR starszych od 4.0 częściowo zaufane zestawy nie mogły nawet wywoływać innych częściowo zaufanych zestawów, jeżeli cel miał silną nazwę (chyba że zastosowano APTCA). Ten środek nie zwiększał jednak poziomu bezpieczeństwa, więc w CLR 4.0 z niego zrezygnowano.

- wywoływać metod oznaczonych w platformie .NET Framework jako [SecurityCritical]; do tej grupy zaliczają się przede wszystkim te metody, które wykonują którąkolwiek z poprzednich czterech czynności bez odpowiednich zabezpieczeń.

Uzasadnienie wprowadzenia tych zasad jest takie, że zestaw, który nie wykonuje żadnej z wymienionych czynności, jest niepodatny na ataki polegające na podnoszeniu uprawnień.

Atrybut [SecurityTransparent] reprezentuje mocniejszą wersję tych samych reguł. Różnica polega na tym, że w przypadku atrybutu APTCA można oznaczać wybrane metody z zestawu jako nietransparentne, natomiast w przypadku atrybutu [SecurityTransparent] wszystkie metody muszą być transparentne.

Jeśli zestaw może działać z atrybutem [SecurityTransparent], to autor biblioteki może uznać swoją pracę za skończoną. Jeśli nie interesują Cię niuanse modelu transparentności, możesz pominąć ten fragment tekstu i przejść od razu do podrozdziału „Zabezpieczenia systemu operacyjnego".

Zanim przyjrzymy się metodom oznaczania wybranych metod jako nietransparentnych, zastanowimy się, w jakich sytuacjach należy w ogóle stosować te atrybuty.

Pierwszy (i bardziej oczywisty) przypadek to sytuacja, gdy planuje się napisanie w pełni zaufanego zestawu, który ma działać w częściowo zaufanej domenie. Konkretny przykład opisaliśmy w sekcji „Ograniczanie innego zestawu".

Drugi (i mniej oczywisty) przypadek jest taki: ktoś pisze bibliotekę, ale nie wie, w jaki sposób będzie ona wdrażana. Powiedzmy np., że piszemy maper obiektowo-relacyjny i sprzedajemy go przez internet. Klienci mogą taką bibliotekę wywoływać na trzy sposoby:

1. w środowisku o pełnym zaufaniu;
2. w domenie ograniczonej;
3. w domenie ograniczonej, ale nadając naszej bibliotece pełne zaufanie (np. przez załadowanie jej do GAC).

Łatwo jest przeoczyć trzecią z tych opcji — i w tej sytuacji pomocny jest model transparentności.

Model transparentności

Najpierw należy przeczytać poprzedni podrozdział, aby wiedzieć, kiedy stosuje się atrybuty APTCA i [SecurityTransparent].

Model transparentności zabezpieczeń ułatwia zabezpieczanie zestawów, które mogą być w pełni zaufane i wywoływane z częściowo zaufanego kodu.

Jako analogiczną sytuację wyobraź sobie, że być częściowo zaufanym zestawem jest jak być skazanym za przestępstwo i wysłanym do więzienia. W więzieniu odkrywamy jednak, że za dobre sprawowanie można zdobyć pewne przywileje (uprawnienia). Dzięki nim możemy robić różne rzeczy, np. oglądać telewizję albo grać w koszykówkę. Ale na pewne rzeczy nigdy nam nie pozwolą, np. nie dostaniemy klucza do pokoju telewizyjnego (ani do bram więzienia), ponieważ takie czynności (metody) podważyłyby cały system zabezpieczeń. Takie metody nazywają się **krytycznymi pod względem zabezpieczeń** (ang. *security-critical*).

Jeśli ktoś pisze w pełni zaufaną bibliotekę, powinien chronić te krytyczne metody. Jednym ze sposobów jest żądanie za pomocą metody `Demand`, aby wywołujący byli w pełni zaufani. Takie podejście stosowano przed pojawieniem się CLR 4.0:

```
[PermissionSet (SecurityAction.Demand, Unrestricted = true)]
public Key GetTVRoomKey() { ... }
```

To stwarza dwa problemy. Przede wszystkim metoda `Demand` jest powolna, ponieważ musi sprawdzać stos wywołań. Ma to znaczenie, ponieważ **metody krytyczne pod względem zabezpieczeń** czasami są też **krytyczne ze względu na wydajność**. Wywołanie metody `Demand` może być szczególnie niekorzystne w przypadku wywoływania metody krytycznej w pętli — np. z innego w pełni zaufanego zestawu platformy. W CLR 2.0 stosowano w takiej sytuacji sztuczkę polegającą na egzekwowaniu **żądań konsolidacji**, w ramach której sprawdzany jest tylko bezpośredni moduł wywołujący. Ale to również ma swoją cenę. Aby utrzymać bezpieczeństwo, metody wywołujące metody z żądaniem konsolidacji same muszą wykonywać żądania uprawnień lub konsolidacji — albo być poddawane kontroli, aby uniemożliwić wykonywanie potencjalnie szkodliwych czynności, jeśli zostaną wywołane z mniej zaufanego modułu. Takie kontrolowanie jest uciążliwe w przypadku skomplikowanych grafów wywołań.

Drugi problem polega na tym, że łatwo zapomnieć wykonać żądania w odniesieniu do metod krytycznych pod względem bezpieczeństwa (komplikują to też skomplikowane grafy wywołań). Byłoby świetnie, gdyby system CLR mógł jakoś pomagać w tym, aby żadne ważne z punktu widzenia bezpieczeństwa funkcje nie zostały przypadkowo udostępnione osobom niepowołanym.

Właśnie to zapewnia model transparentności.

Wprowadzenie modelu transparentności nie ma żadnego związku z usunięciem **zasady CAS** (zob. ramka „Zasady zabezpieczeń w CLR 2.0").

Zasada działania modelu transparentności

W modelu transparentności metody krytyczne pod względem bezpieczeństwa są oznaczone atrybutem `[SecurityCritical]`:

```
[SecurityCritical]
public Key GetTVRoomKey() { ... }
```

Wszystkie „niebezpieczne" metody (także te, które wg CLR mogą złamać zabezpieczenia i umożliwić ucieczkę więźniom) muszą być oznaczone atrybutem `[SecurityCritical]` lub `[SecuritySafeCritical]`. Do tych metod zaliczają się:

- metody nieweryfikowalne (niebezpieczne);
- metody wywołujące kod niezarządzany przez P/Invoke lub COM;
- metody żądające uprawnień za pomocą metody `Assert` lub wywołujące metody żądające konsolidacji;
- metody *wywołujące* metody oznaczone atrybutem `[SecurityCritical]`;
- metody *przesłaniające* wirtualne metody z atrybutem `[SecurityCritical]`.

Obecność atrybutu `[SecurityCritical]` oznacza, że metoda może umożliwić częściowo niezaufanemu wywołującemu ucieczkę ze środowiska ograniczonego.

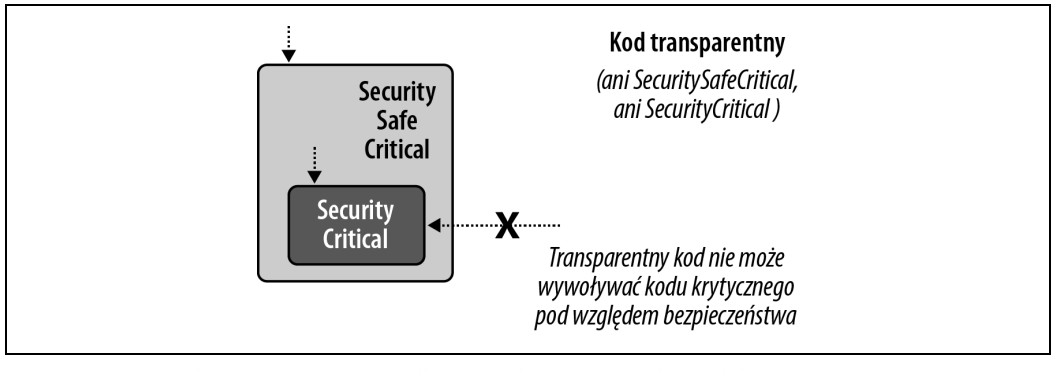
Obecność atrybutu `[SecuritySafeCritical]` oznacza, że metoda wykonuje czynności o krytycznym znaczeniu z punktu widzenia bezpieczeństwa — ale ma wbudowane odpowiednie zabezpieczenia, więc może być bezpiecznie używana przez częściowo zaufanych wywołujących.

Metody znajdujące się w częściowo zaufanych zestawach nie mogą wywoływać metod krytycznych pod względem bezpieczeństwa z w pełni zaufanych zestawów. Metody z atrybutem `[SecurityCritical]` mogą być wywoływane tylko przez:

- inne metody z atrybutem `[SecurityCritical]`,
- metody oznaczone jako `[SecuritySafeCritical]`.

Bezpieczne metody krytyczne działają jak strażnicy dla metod krytycznych pod względem bezpieczeństwa (rysunek 21a.1) i mogą być wywoływane przez każdą metodę znajdującą się w dowolnym zestawie (w pełni lub częściowo zaufanym, zgodnie z żądaniami CAS). Powiedzmy np., że jako osadzeni chcemy oglądać telewizję. Metoda `WatchTV`, którą wywołamy, będzie musiała wywołać metodę `GetTVRoomKey`, co oznacza, że metoda `WatchTV` musi być *bezpieczną metodą krytyczną*:

```
[SecuritySafeCritical]
public void WatchTV()
{
  new TVPermission().Demand();
  using (Key key = GetTVRoomKey())
    PrisonGuard.OpenDoor (key);
}
```



Rysunek 21a.1. Model transparentności. Tylko szary obszar wymaga kontroli bezpieczeństwa

Zwróć uwagę, że żądamy, aby metoda `TVPermission` sprawdzała, czy wywołujący ma uprawnienia do oglądania telewizji, oraz kasujemy utworzony przez nas klucz. Opakowaliśmy metodę *krytyczną pod względem bezpieczeństwa*, czyniąc ją *bezpieczną* do wywołania dla każdego.

Niektóre metody biorą udział w czynnościach, które są uważane za „niebezpieczne" przez CLR, ale w rzeczywistości takie nie są. Można je bezpośrednio oznaczyć za pomocą atrybutu `[SecuritySafeCritical]` zamiast `[SecurityCritical]`. Dobrym przykładem jest metoda `Array.Copy` — istnieje jej niezarządzana implementacja o podwyższonej efektywności, której nie da się wykorzystać do złych celów w częściowo zaufanym kodzie.

Wzorzec UnsafeXXX

W naszym przykładzie dotyczącym telewizji istnieje ryzyko pogorszenia wydajności polegające na tym, że jeżeli strażnik więzienny zechce obejrzeć telewizję i w tym celu wywoła metodę `WatchTV`, będzie musiał (niepotrzebnie) zaspokoić żądanie `TVPermission`. W ramach rozwiązania zespół ds. CLR zaleca stosowanie wzorca, wg którego definiuje się dwie wersje metody. Pierwsza jest krytyczna pod względem bezpieczeństwa i ma w nazwie przedrostek **`Unsafe`**:

```
[SecurityCritical]
public void UnsafeWatchTV()
{
  using (Key key = GetTVRoomKey())
    PrisonGuard.OpenDoor(key);
}
```

Druga z kolei to metoda `SecuritySafeCritical`, która wywołuje pierwszą, gdy spełni warunek nakazujący jej przejrzenie całego stosu wywołań:

```
[SecuritySafeCritical]
public void WatchTV()
{
  new TVPermission().Demand();
  UnsafeWatchTV();
}
```

Kod transparentny

W modelu transparentności wszystkie metody przypadają do jednej z trzech kategorii:

- *bezpieczne pod względem zabezpieczeń*;
- *bezpieczne-krytyczne pod względem zabezpieczeń*;
- żadne z powyższych (i wówczas nazywają się *transparentne*).

Nazwa **metoda transparentna** odnosi się do tego, że taką metodę można zignorować podczas sprawdzania, czy w kodzie nie ma nielegalnych prób podnoszenia uprawnień. Programista musi tylko się skupić na metodach `[SecuritySafeCritical]` (strażnikach), które z reguły stanowią niewielki ułamek wszystkich metod znajdujących się w zestawie. Jeżeli zestaw zawiera same metody transparentne, to może być w całości oznaczony atrybutem `[SecurityTransparent]`:

```
[assembly: SecurityTransparent]
```

Wówczas *sam zestaw* jest transparentny. Takie zestawy nie muszą być sprawdzane pod kątem znamion ataków podnoszenia uprawnień i niejawnie dopuszczają częściowo zaufanych wywołujących — nie trzeba stosować atrybutu APTCA.

Domyślne ustawienia przezroczystości zestawu

Podsumowując wcześniejsze wywody, można wyróżnić dwa sposoby na określenie transparentności na poziomie zestawu:

- przez zastosowanie atrybutu APTCA — domyślnie transparentne są wszystkie te metody, których specjalnie nie oznaczono inaczej;
- przez zastosowanie atrybutu `[SecurityTransparent]` — wówczas transparentne są wszystkie metody bez wyjątku.

Istnieje też trzecia możliwość polegająca na nierobieniu niczego. Wtedy nadal obowiązują zasady transparentności, ale każda metoda jest domyślnie `[SecurityCritical]` (nie licząc wirtualnych metod `[SecuritySafeCritical]`, które zostaną przesłonięte i pozostaną bezpieczne-krytyczne). W efekcie można wywołać każdą metodę (przy założeniu, że ma się pełne zaufanie), ale metody transparentne z innych zestawów nie będą mogły wywoływać nas.

Jak pisać biblioteki APTCA z transparentnością

Aby spełnić wymagania modelu transparentności, najpierw należy w zestawie zidentyfikować potencjalnie „niebezpieczne" metody (zgodnie z opisem zamieszczonym w poprzedniej sekcji). Można do tego celu wykorzystać testy jednostkowe, ponieważ CLR odmówi wykonania takich metod — nawet we w pełni zaufanym środowisku. (Platforma .NET Framework zawiera też pomocne narzędzie o nazwie *SecAnnotate.exe*). Następnie każdą metodę należy oznaczyć jednym z dwóch atrybutów:

- `[SecurityCritical]` — jeśli metoda może być szkodliwa, gdy zostanie wywołana w mniej zaufanym zestawie.
- `[SecuritySafeCritical]` — jeśli metoda wykonuje odpowiednie testy i ma zabezpieczenia, dzięki którym można ją bezpiecznie wywoływać w mniej zaufanym zestawie.

Spójrz np. na poniższą metodę, która wywołuje metodę krytyczną pod względem bezpieczeństwa na platformie .NET Framework:

```
public static void LoadLibraries()
{
  GC.AddMemoryPressure (1000000); // SecurityCritical
  ...
}
```

Mniej zaufane moduły mogłyby narobić szkód, wywołując tę metodę wielokrotnie. Można by było dodać atrybut `[SecurityCritical]`, ale wówczas metodę można by było wywoływać tylko z innych zaufanych modułów poprzez krytyczne lub bezpieczne-krytyczne metody. Lepszym rozwiązaniem jest więc takie poprawienie metody, aby była bezpieczna, i dodanie jej atrybutu `[SecuritySafeCritical]`:

```
static bool _loaded;

[SecuritySafeCritical]
public static void LoadLibraries()
{
  if (_loaded) return;
  _loaded = true;
  GC.AddMemoryPressure (1000000);
  ...
}
```

(Zaletą tego rozwiązania jest to, że metoda staje się bezpieczniejsza także dla zaufanych wywołujących).

Zabezpieczanie niebezpiecznych metod

Teraz powiedzmy, że mamy niebezpieczną (`unsafe`) metodę, która może być szkodliwa, jeśli zostanie wywołana przez mniej zaufany zestaw. Taką metodę należy po prostu opatrzyć atrybutem `[SecurityCritical]`:

```
[SecurityCritical]
public unsafe void Poke (int offset, int data)
{
  int* target = (int*) _origin + offset;
  *target = data;
  ...
}
```

Jeśli w metodzie transparentnej znajdzie się niebezpieczny kod, system CLR przed wykonaniem metody zgłosi wyjątek `VerificationException` („Operacja może spowodować niestabilne działanie w czasie wykonywania").

Następnie zabezpieczamy metody nadrzędne, oznaczając je zgodnie z potrzebą atrybutami `[Security↪Critical]` i `[SecuritySafeCritical]`.

Teraz spójrz na poniższą metodę niebezpieczną, która filtruje mapy bitowe. Jest ona nieszkodliwa z natury, więc możemy ją oznaczyć atrybutem `[SecuritySafeCritical]`.

```
[SecuritySafeCritical]
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Da się też napisać metodę, która będzie „nieszkodliwa" z punktu widzenia CLR, a w rzeczywistości będzie stwarzała niebezpieczeństwo. Taką metodę można oznaczyć atrybutem `[SecurityCritical]`:

```
public string Password
{
  [SecurityCritical] get { return _password; }
}
```

P/Invoke i atrybut [SuppressUnmanagedSecurity]

Na koniec spójrz na poniższą niezarządzaną metodę, która zwraca uchwyt do okna z obiektu klasy `Point` (`System.Drawing`):

```
[DllImport ("user32.dll")]
public static extern IntPtr WindowFromPoint (Point point);
```

Przypominamy, że kod niezarządzany można wywoływać tylko w metodach `[SecurityCritical]` i `[SecuritySafeCritical]`.

Można powiedzieć, że wszystkie metody `extern` są domyślnie oznaczone jako `[Security↪Critical]`, choć nie zachowują się dokładnie w taki sposób — bezpośrednie przypisanie atrybutu `[SecurityCritical]` takiej metodzie powoduje przeniesienie procesu sprawdzania bezpieczeństwa z czasu wykonywania do JIT. Spójrz np. na poniższą metodę:

```
static void Foo (bool exec)
{
  if (exec) WindowFromPoint (...)
}
```

Gdyby w jej wywołaniu przekazano wartość `false`, to podlegałoby ono pod test bezpieczeństwa tylko, gdyby metoda `WindowFromPoint` była jawnie oznaczona atrybutem `[SecurityCritical]`.

Jako że nasza metoda jest publiczna, inne w pełni zaufane zestawy także mogą wywoływać metodę `WindowFromPoint` bezpośrednio w metodach `[SecurityCritical]`. W przypadku modułów o częściowym zaufaniu udostępniamy poniższą bezpieczną wersję, w której wyeliminowano niebezpieczeństwo przez zażądanie uprawnienia `UIPermission` i zwrot klasy zarządzanej zamiast `IntPtr`:

```
[UIPermission (SecurityAction.Demand, Unrestricted = true)]
[SecuritySafeCritical]
public static System.Windows.Forms.Control ControlFromPoint (Point point)
{
  IntPtr winPtr = WindowFromPoint (point);
  if (winPtr == IntPtr.Zero) return null;
  return System.Windows.Forms.Form.FromChildHandle (winPtr);
}
```

Pozostaje jeszcze tylko jeden problem — system CLR wykonuje niejawne żądanie niezarządzanego uprawnienia, gdy używane są usługi P/Invoke. A ponieważ metoda `Demand` sprawdza cały stos w górę, wykonanie metody `WindowFromPoint` się nie uda, jeżeli moduł nadrzędny wywołującego będzie częściowo zaufany. Można to obejść na dwa sposoby. Pierwszym jest *żądanie* uprawnienia dla niezarządzanego kodu w pierwszym wierszu metody `ControlFromPoint`:

```
new SecurityPermission (SecurityPermissionFlag.UnmanagedCode).Assert();
```

Żądanie prawa do wykonywania niezarządzanego kodu w tym przypadku sprawi, że następne niejawne wywołanie metody `Demand` w `WindowFromPoint` będzie udane. Oczywiście nie powiodłoby się to, gdyby sam zestaw nie był w pełni zaufany (dzięki temu, że jest ładowany do GAC lub oznaczony jako w pełni zaufany przez hosta). Szerzej o asercjach piszemy w sekcji „Ograniczanie innego zestawu”.

Drugie (efektywniejsze) rozwiązanie polega na dodaniu atrybutu [SuppressUnmanagedCodeSecurity] do niezarządzanej metody:

```
[DllImport ("user32.dll"), SuppressUnmanagedCodeSecurity]
public static extern IntPtr WindowFromPoint (Point point);
```

Atrybut ten nakazuje systemowi CLR zaniechanie czasochłonnego przeglądania stosu w wyniku wywołania metody Demand (taka optymalizacja może być szczególnie cenna, jeśli metoda WindowFrom
↪Point będzie wywoływana z innych zaufanych klas lub zestawów). Następnie możemy wrzucić żądanie uprawnienia do wykonywania kodu niezarządzanego do metody ControlFromPoint.

Postępujemy zgodnie z modelem transparentności, więc zastosowanie tego atrybutu do metody extern nie stwarza takiego samego ryzyka jak w CLR 2.0. Przyczyną tego jest fakt, że nadal chroni nas to, że usługi P/Invoke niejawnie są krytyczne pod względem bezpieczeństwa, więc mogą być wywoływane tylko z metod krytycznych lub bezpiecznych-krytycznych.

Transparentność w przypadku pełnego zaufania

We w pełni zaufanym środowisku można napisać kod krytyczny i jednocześnie starać się uniknąć stosowania atrybutów zabezpieczeń i audytów metod. Najprostszym sposobem na osiągnięcie tego celu jest niedodawanie żadnych atrybutów dotyczących bezpieczeństwa zestawu — w takim przypadku wszystkie metody są domyślnie [SecurityCritical].

Taka metoda się sprawdza, dopóki *wszystkie* interesujące nas zestawy robią to samo — lub jeśli zestawy transparentne znajdują się na *dole* grafu wywołań. Innymi słowy: w bibliotekach zewnętrznych (i .NET Framework) także można wywoływać metody transparentne.

Problemy można mieć natomiast w odwrotnej sytuacji, choć często prowadzą one do znalezienia lepszego rozwiązania. Powiedzmy, że piszemy zestaw T, który jest częściowo lub całkowicie transparentny, i chcemy wywołać zestaw X, który nie ma żadnych atrybutów (a więc cały jest krytyczny). Do wyboru mamy trzy możliwości:

- Zamienienie własnego zestawu w całkowicie krytyczny. Jeśli nasza domena będzie zawsze w pełni zaufana, nie musimy obsługiwać częściowo zaufanych wywołujących. Ukazanie *wprost* tego braku obsługi ma sens.
- Napisanie opakowań [SecuritySafeCritical] dla metod w zestawie X. W ten sposób wskazuje się czułe punkty zabezpieczeń (choć może to być kłopotliwe do wykonania).
- Podsunięcie twórcy zestawu X pomysłu, aby rozważyć transparentność. Jeśli zestaw X nie robi niczego krytycznego, wystarczy tylko dodać do zestawu X atrybut [SecurityTransparent]. Jeśli natomiast zestaw X wykonuje pewne krytyczne funkcje, autor zestawu będzie musiał przynajmniej znaleźć (i może wyeliminować) czułe punkty.

Zasady zabezpieczeń w CLR 2.0

W wersjach starszych niż 4.0 system CLR przyznawał domyślny zestaw uprawnień zestawom .NET na podstawie skomplikowanych zbiorów reguł i mapowań. Nazywało się to **zasadami** CAS, które były zdefiniowane w konfiguracji platformy .NET Framework komputera. Z tych zasad powstały trzy standardowe zbiory uprawnień, które można dostosowywać na poziomie domeny przedsiębiorstwa, maszyny, użytkownika i aplikacji:

- pełne zaufanie — przydzielany zestawom działającym na lokalnym dysku twardym;
- lokalny intranet — przydzielany zestawom działającym w udziale sieciowym;
- internet — przydzielany zestawom działającym w Internet Explorerze.

Domyślnie pełne zaufanie zapewniał tylko pierwszy z tych trzech zbiorów. Oznaczało to, że jeśli ktoś uruchomił aplikację .NET na udziale sieciowym, mogła ona mieć ograniczone uprawnienia, co często wiązało się z nieudanym uruchomieniem. Chodziło o zapewnienie ochrony, ale w rzeczywistości nie udało się nic osiągnąć, ponieważ mający złe zamiary użytkownik mógł podmienić plik wykonywalny .NET na niezarządzany plik wykonywalny i ominąć wszelkie ograniczenia dotyczące uprawnień. Jedynym osiągnięciem tych zabezpieczeń było zdenerwowanie ludzi, którzy chcieli uruchamiać zestawy .NET z pełnym zaufaniem przez udziały sieciowe.

Dlatego projektanci CLR 4.0 postanowili anulować opisane zasady bezpieczeństwa. Teraz wszystkie zestawy działają ze zbiorem uprawnień zdefiniowanym w całości przez środowisko hosta. Pliki wykonywalne, które uruchamia się za pomocą dwukrotnego kliknięcia lub w wierszu poleceń zawsze mają pełne zaufanie — niezależnie od tego, czy znajdują się w udziale sieciowym, czy na lokalnym dysku twardym.

Innymi słowy, to *od hosta zależy*, jak powinny być ograniczone uprawnienia — zasady CAS komputera nie mają na to wpływu.

Ograniczanie innego zestawu

Przyjmijmy, że piszemy aplikację, w której można instalować zewnętrzne wtyczki. Oczywiście te wtyczki nie powinny mieć możliwości korzystania z takich samych uprawnień, jakie ma nasza zaufana aplikacja, aby zapobiec ewentualnej destabilizacji programu lub komputera użytkownika. W takiej sytuacji najlepszym rozwiązaniem jest uruchamianie każdej wtyczki w osobnej ograniczonej domenie.

Poniżej przedstawiamy przykład, w którym wtyczka jest zestawem .NET o nazwie *plugin.exe*, który można aktywować poprzez jego uruchomienie. (W rozdziale 24. opisujemy sposoby ładowania bibliotek do domeny aplikacji i posługiwania się nimi w bardziej wyszukane sposoby).

Oto kompletny kod programu *hosta*:

```
using System;
using System.IO;
using System.Net;
using System.Reflection;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;

class Program
{
  static void Main()
```

```
  {
    string pluginFolder = Path.Combine (
      AppDomain.CurrentDomain.BaseDirectory, "plugins");

  string plugInPath = Path.Combine (pluginFolder, "plugin.exe");

  PermissionSet ps = new PermissionSet (PermissionState.None);

  ps.AddPermission
    (new SecurityPermission (SecurityPermissionFlag.Execution));

    ps.AddPermission
      (new FileIOPermission (FileIOPermissionAccess.PathDiscovery |
                             FileIOPermissionAccess.Read, plugInPath));

    ps.AddPermission (new UIPermission (PermissionState.Unrestricted));

    AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
    AppDomain sandbox = AppDomain.CreateDomain ("sbox", null, setup, ps);
    sandbox.ExecuteAssembly (plugInPath);
    AppDomain.Unload (sandbox);
  }
}
```

W razie potrzeby do metody `CreateDomain` można przekazać tablicę obiektów typu `StrongName` wskazujących zestawy mające pełne zaufanie. Przykład takiego rozwiązania przedstawiamy w następnej sekcji.

Najpierw utworzymy ograniczony zbiór uprawnień, jaki chcemy przydzielić naszemu ograniczonemu środowisku. Musi on obejmować przynajmniej prawo do wykonywania programów oraz możliwość wczytywania przez wtyczki ich własnych zestawów — inaczej wtyczek nie da się uruchamiać. Dodatkowo w tym przypadku dodamy też nieograniczone uprawnienia dotyczące interfejsu użytkownika. Następnie utworzymy nową domenę aplikacji, której przydzielimy nasz zbiór uprawnień, aby były dostępne dla wszystkich ładowanych do niej zestawów. Później uruchomimy zestaw wtyczki w tej domenie oraz skasujemy tę domenę, gdy wtyczka zakończy działanie.

W tym przykładzie zestawy wtyczki wczytujemy z podkatalogu o nazwie *plugins*. Należy jednak wiedzieć, że obecność wtyczek w katalogu mającego pełne zaufanie hosta wystawia system na ryzyko ataku podniesienia uprawnień polegającego na tym, że w pełni zaufana domena niejawnie wczyta i wykona kod z zestawu wtyczki w celu rozpoznania typu. Do takiej sytuacji może dojść np. wtedy, gdy wtyczka zgłosi własny typ wyjątku, który jest zdefiniowany w jej zestawie. Kiedy ten wyjątek drogą propagacji dojdzie do hosta, ten wczyta zestaw wtyczki, jeśli go znajdzie — aby dokonać deserializacji wyjątku. Gdyby umieścić wtyczki w osobnym folderze, taka sztuczka nie mogłaby się udać.

Żądanie uprawnień

Uprawnień można żądać w metodach, które będą wywoływane z częściowo zaufanych zestawów. W ten sposób umożliwia się w pełni zaufanym zestawom tymczasowe wyjście z ograniczonego środowiska w celu wykonania czynności, których bez tego nie dałoby się przeprowadzić z powodu dalszych wywołań metody `Demand`.

Żądania (asercje) uprawnień w świecie CAS nie mają nic wspólnego z asercjami diagnostycznymi ani kontraktowymi. Wywołanie metody `Debug.Assert` jest nawet bliższe wywołaniu metody `Demand` niż `Assert` w celu zdobycia uprawnień. Przede wszystkim operacja żądania uprawnień ma *skutek uboczny*, jeśli żądanie się powiedzie. Natomiast metoda `Debug.Assert` nie ma skutków ubocznych.

Wcześniej napisaliśmy aplikację uruchamiającą wtyczki z ograniczonym zbiorem uprawnień. Teraz powiedzmy, że chcemy dostarczyć bibliotekę bezpiecznych metod, które te wtyczki będą mogły wywoływać. Moglibyśmy np. uniemożliwić wtyczkom bezpośrednie odwoływanie się do bazy danych, ale pozostawić furtkę w postaci kilku metod wykonujących wybrane rodzaje zapytań. Albo moglibyśmy udostępnić metodę zapisującą dane w pliku dziennika, unikając w ten sposób nadawania wtyczkom uprawnień do zapisu w plikach.

Pierwszą naszą czynnością będzie zatem utworzenie osobnego zestawu (np. *utilities*) i dodanie atrybutu `AllowPartiallyTrustedCallers`. Dzięki temu będzie można udostępniać metody w następujący sposób:

```
public static void WriteLog (string msg)
{
  // zapis do dziennika
  ...
}
```

Trudność w tym przypadku polega na tym, że aby móc zapisać cokolwiek w pliku, należy mieć uprawnienie `FileIOPermission`. I choć nasz zestaw *utilities* będzie miał pełne zaufanie, moduł z niego korzystający już nie, przez co wszelkie żądania `Demand` dotyczące operacji na plikach zakończą się fiaskiem. Rozwiązaniem jest uprzednie zażądanie odpowiedniego uprawnienia za pomocą metody `Assert`:

```
public class Utils
{
  string _logsFolder = ...;

  [SecuritySafeCritical]
  public static void WriteLog (string msg)
  {
    FileIOPermission f = new FileIOPermission (PermissionState.None);
    f.AddPathList (FileIOPermissionAccess.AllAccess, _logsFolder);
    f.Assert();

    // zapis do dziennika
    ...
  }
}
```

Żądanie uprawnień w metodzie oznacza, że trzeba do niej dodać atrybut `[Security↪Critical]` lub `[SecuritySafeCritical]` (chyba że interesuje nas starsza wersja platformy). Nasza metoda jest bezpieczna dla częściowo zaufanych modułów wywołujących, więc wybieramy atrybut `SecuritySafeCritical`. To oczywiście powoduje, że nie możemy oznaczyć całego zestawu za pomocą atrybutu `[SecurityTransparent]`, tylko musimy użyć APTCA.

Przypominamy, że metoda Demand wykonuje test w miejscu wywołania i zgłasza wyjątek, jeśli dane uprawnienie jest niedostępne. Następnie przegląda stos, aby sprawdzić, czy wszystkie nadrzędne moduły wywołujące również mają potrzebne uprawnienie (w obrębie bieżącej domeny aplikacji). Asercja sprawdza tylko, czy *bieżący zestaw* ma odpowiednie uprawnienia, i jeśli wynik testu jest pozytywny, wprowadza na stos znacznik sygnalizujący, że od tego miejsca prawa modułu wywołującego mają być ignorowane, a pod uwagę mają być brane tylko prawa bieżącego zestawu w odniesieniu do tamtych uprawnień. Koniec asercji następuje wraz z zakończeniem działania metody lub po wywołaniu metody CodeAccessPermission.RevertAssert.

Ostatnim krokiem w naszym przykładzie jest utworzenie ograniczonej domeny aplikacji, która w pełni ufa zestawowi *utilities*. Później będziemy mogli utworzyć obiekt typu StrongName opisujący ten zestaw i przekazać go do metody CreateDomain obiektu klasy AppDomain:

```
static void Main()
{
  string pluginFolder = Path.Combine (
    AppDomain.CurrentDomain.BaseDirectory, "plugins");

  string plugInPath = Path.Combine (pluginFolder, "plugin.exe");

  PermissionSet ps = new PermissionSet (PermissionState.None);

  // dodanie potrzebnych uprawnień do ps, jak wcześniej
  // ...

  Assembly utilAssembly = typeof (Utils).Assembly;
  StrongName utils = utilAssembly.Evidence.GetHostEvidence<StrongName>();

  AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
  AppDomain sandbox = AppDomain.CreateDomain ("sbox", null, setup, ps,
                                              utils);
  sandbox.ExecuteAssembly (plugInPath);
  AppDomain.Unload (sandbox);
}
```

Aby ten kod zadziałał, zestaw *utilities* musi być podpisany przy użyciu silnej nazwy.

W wersjach platformy starszych niż 4.0 nie było możliwości utworzenia obiektu StrongName za pomocą wywołania metody GetHostEvidence, jak to zrobiliśmy w powyższym przykładzie. Zamiast tego stosowano poniższe rozwiązanie:

```
AssemblyName name = utilAssembly.GetName();
StrongName utils = new StrongName (
  new StrongNamePublicKeyBlob (name.GetPublicKey()),
  name.Name,
  name.Version);
```

Rozwiązanie to można nadal stosować, gdy nie ładuje się zestawu do domeny hosta. Jest to możliwe, ponieważ obiekt typu AssemblyName można utworzyć bez użycia obiektów typu Assembly czy Type:

```
AssemblyName name = AssemblyName.GetAssemblyName
                    (@"d:\utils.dll");
```