KAPITOLA 18

Jak psát bezpečný kód .NET

Tuto kapitolu musím zahájit menším příběhem. Když jsem si někdy v listopadu 2001 připravoval slajdy ke dvěma přednáškám o bezpečnosti softwaru na Microsoft Professional Developer's Conference, řekl mi jeden známý, že brzo určitě přijdu o práci, protože jakmile se na trhu objeví řízený kód (Managed C) a technologie .NET Framework, veškeré bezpečnostní problémy tím provždy zmizí. Na základě tohoto tvrzení jsem upravil svou ukázku injekce kódu SQL (rozebírali jsme ji v kapitole 12) z původního C++ na C# – a ukázal jsem mu, jak hluboce se mýlí.

Řízený kód jistě zvládne alespoň část bezpečnostních problémů za vývojáře, zejména pokud umíte programovat v jazyce C nebo C++, ale „vypnout mozek" zkrátka nemůžete nikdy, ať už se pohybujete v jakémkoli programovacím jazyce. Věřím, že si vezmete k srdci veškeré zásady ohledně správného návrhu a kódování i z této kapitoly a že i první aplikaci .NET budete tvořit v souladu s nimi. Říkám to z toho důvodu, že jsme dnes na vrcholu rozšíření Microsoft .NET, a čím dříve se nám podaří posílit celkové povědomí a čím více programátorů či vývojářů naučíme vytvářet bezpečný software hned od začátku, tím lépe bude pro všechny. V této kapitole si řekneme něco o různých bezpečnostních chybách, kterým se můžeme vyhnout, a také o jistých doporučených postupech, kterých je dobré se držet při psaní kódu pod společnou běhovou knihovnou .NET CLR (Common Language Runtime), webových služeb i XML.

Nezapomeňte také, že řízeného kódu se týká i množství různých ponaučení ve zbytku knížky; jako příklady mohu jmenovat:

- Neukládejte tajné informace do kódu ani do souborů *web.config*.
- Nepokoušejte se vytvářet svoje vlastní šifrování, ale využívejte raději třídy v oboru názvů `System.Security.Cryptography`.
- Nedůvěřujte žádnému vstupu, dokud pozitivně neověříte jeho správnost.

Řízený kód, který je zajištěn ve společné běhové knihovně .NET, nám pomůže potlačit celou řadu zranitelných míst v zabezpečení (například přetečení bufferu) a také některé problémy spojené s plně důvěryhodným mobilním kódem, jako jsou ovládací prvky ActiveX. V rámci tradičního zabezpečení v Microsoft Windows se ovšem při bezpečnostních kontrolách uvažuje jen identita nositele zabezpečení (hlavního objektu, principal). Jinými slovy, pokud je daný uživatel důvěryhodný, běží příslušný kód pod jeho identitou, a je tedy také stejně důvěryhodný a má stejná oprávnění jako uživatel. Pod Windows 2000 a novějšími je k dispozici technologie založená na omezených tokenech, která uvedené problémy zčásti potlačuje; podrobnější informace k omezeným tokenům jsou uvedeny v kapitole 7. Bezpečnost v systému .NET je však ještě na vyšší úrovni a kód v něm definuje různé úrovně důvěry nejen podle toho, jaké možnosti má uživatel v systému, ale také podle systémové politiky (zásad) a podle takzvané *legitimace* kódu (evidence, tedy „důkazů" o kódu). Mezi tyto legitimace patří různé vlastnosti kódu, jako je digitální podpis nebo síť jeho původu, na základě kterých tato bezpečnostní politika přidělí kódu potřebná oprávnění.

To je hodně důležité, protože zejména ve světě propojeném do sítě Internet chtějí uživatelé často spouštět kód od neznámých autorů, a nemají naprosto žádnou záruku, že byl napsán bezpečně. Pokud budeme důvěřovat kódu méně než uživateli (což je jen jedna z možných kombinací důvěry k uživateli a důvěry ke kódu), mohou vysoce důvěryhodní uživatelé spouštět kód i bez nežádoucího rizika. Nejběžnějším příkladem z dnešní doby jsou skripty na webových stránkách: skript může bezpečně přijít z libovolného webového serveru (za předpokladu, že je bezpečná i implementace prohlížeče), protože okruh funkcí povolených ve skriptu je přísně omezený. Bezpečnost systému .NET dále zobecňuje pojem důvěry ke kódu a nabízí mnohem silnější kompromisy mezi bezpečností a funkčností; důvěra je přitom založena jen na legitimacích, nikoli na pevném, předem určeném modelu jako u webového skriptu.

Poznámka: Nejlepší a nejbezpečnější jsou podle mého názoru takové aplikace, které využívají těch nejlepších bezpečnostních funkcí operačního systému a zároveň i nejlepších bezpečnostních funkcí .NET, protože každý z nich řeší problémy zabezpečení z jiného pohledu. Ani jeden z těchto pohledů není univerzálním všelékem, a proto je důležité vědět, které technologie jsou při výstavbě konkrétní aplikace nejvhodnější. A o těchto nejvhodnějších technologiích rozhodneme na základě modelu hrozeb.

Nenechte se ale v žádném případě ukolébat falešným pocitem bezpečí. Architektura .NET i řízený kód nabízí sice množství různých postupů pro snížení pravděpodobnosti určitých útoků, ale úplně před vším nás nikdo ochránit nedokáže.

Důležité: Společná běhová knihovna CLR se umí bránit proti jistým typům bezpečnostních chyb, ale to samozřejmě neznamená, že bychom mohli líně „usnout na vavřínech". Pokud nebudeme dodržovat základní bezpečnostní pravidla, nepomohou nám ani ty nejlepší bezpečnostní funkce.

Než se pustíme do slíbených nejlepších doporučených postupů, podíváme se krátce do světa bezpečnosti kódu pro přístup v .NET, Code Access Security (CAS).

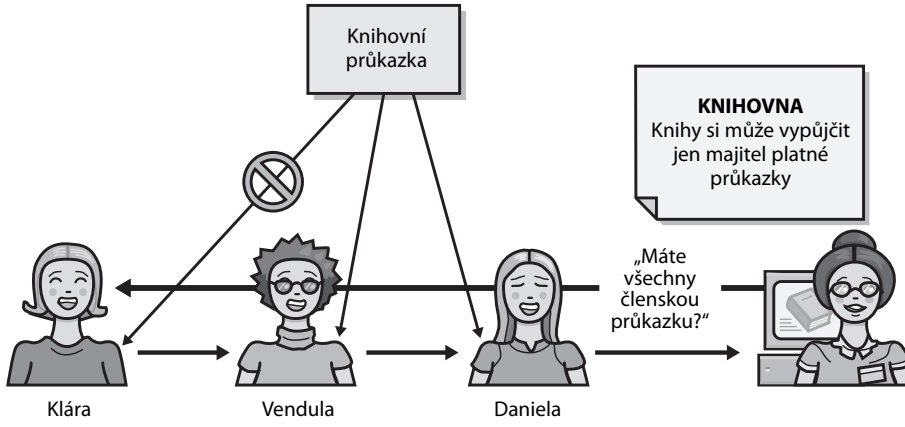
Bezpečnost kódu pro přístup: obrazem

V této části textu si stručně načrtneme základní elementy bezpečnosti kódu pro přístup (Code Access Security, CAS) v běhové knihovně .NET CLR. Výklad je jen rámcový a rozhodně nemá za cíl nahradit podrobné, hloubkové vysvětlení celé problematiky, jaké najdete například v publikaci *.NET Framework Security* (viz odkaz v doporučené literatuře), ale mělo by z něj být jasné, jak CAS funguje, a zároveň byste z něj měli pochopit některé základní pojmy používané ve zbytku kapitoly.

Nebudu zde zacházet do přílišných podrobností a scénář podobný přístupovému zabezpečení CAS přiblížím raději na názorných diagramech, které hovoří o praktickém příkladu zapůjčení knížky z knihovny. V našem příkladu si Klára bude chtít vypůjčit knihu z knihovny, ale protože nemá členskou průkazku, požádá o vypůjčení své dvě kamarádky, Vendulu a Danielu. Podívejte se na obrázek 18.1.

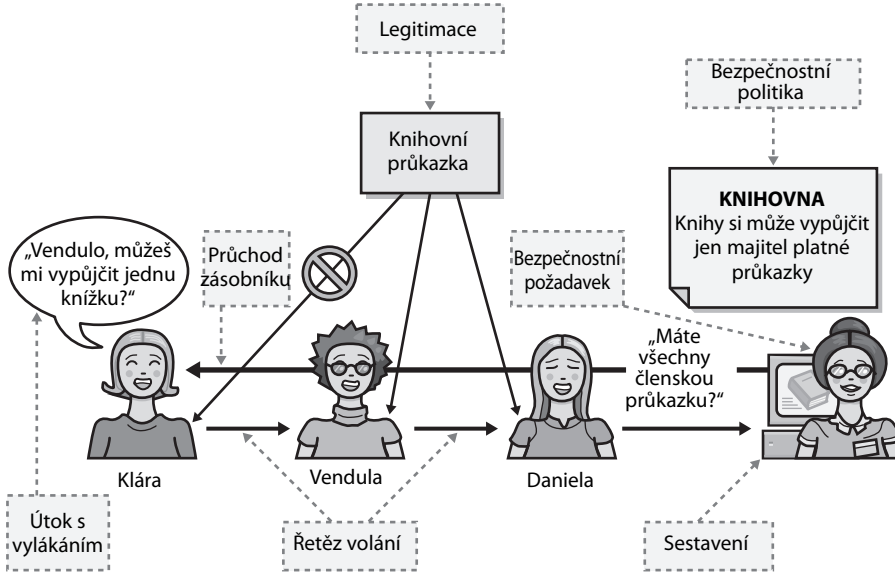
Obrázek 18.1 Klára si chce vypůjčit knihu z knihovny a požádá přitom své dvě kamarádky

Život ale ve skutečnosti není tak jednoduchý; konec konců, kdyby knihovníci půjčovali knihy komukoli z ulice, za chvíli by v policích neměli nic. Knihy je proto nutné chránit nějakou bezpečnostní či „přístupovou politikou" – knihovna je půjčuje jen držitelům knihovní průkazky. Jak ale vidíme z obrázku 18.2, Klára takovou průkazku nemá.



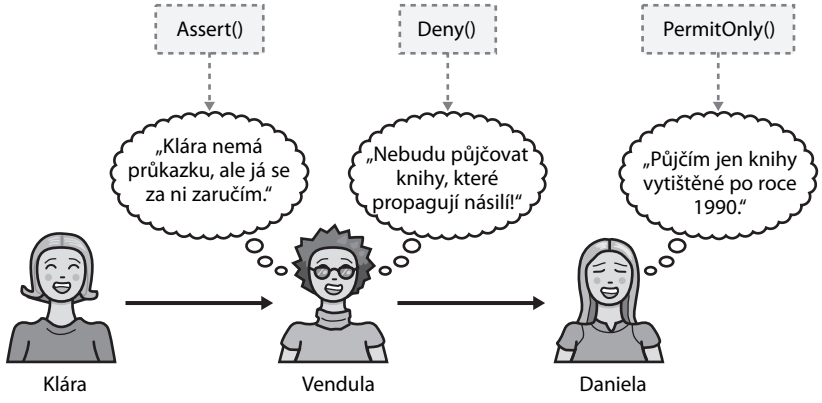
Obrázek 18.2 Platí-li „přístupová politika“ knihovny, nemůže si Klára knihu vypůjčit, protože nemá průkazku

Věřte nebo nevěřte – ale právě jste se naučili základy přístupové bezpečnosti, CAS! Podívejme se na celý scénář ještě jednou a doplňme do něj terminologii CAS; začneme obrázkem 18.3.



Obrázek 18.3 Zajištění platnosti „přístupové politiky“ knihovny – podle pojmů CAS

V reálném světě se asi najde hodně způsobů, jak celý systém „rozvolnit“ a jak nakonec Kláře knihu zapůjčit, ale musí k tomu být splněny jisté podmínky, které si definuje Vendula a Daniela. Celý scénář opět mírně přepíšeme a dostaneme se k obrázku 18.4, kde jsou již jisté modifikátory z přístupové bezpečnosti CAS.



Obrázek 18.4 Jak se požadavky z reálného světa promítají do zabezpečovacího systému

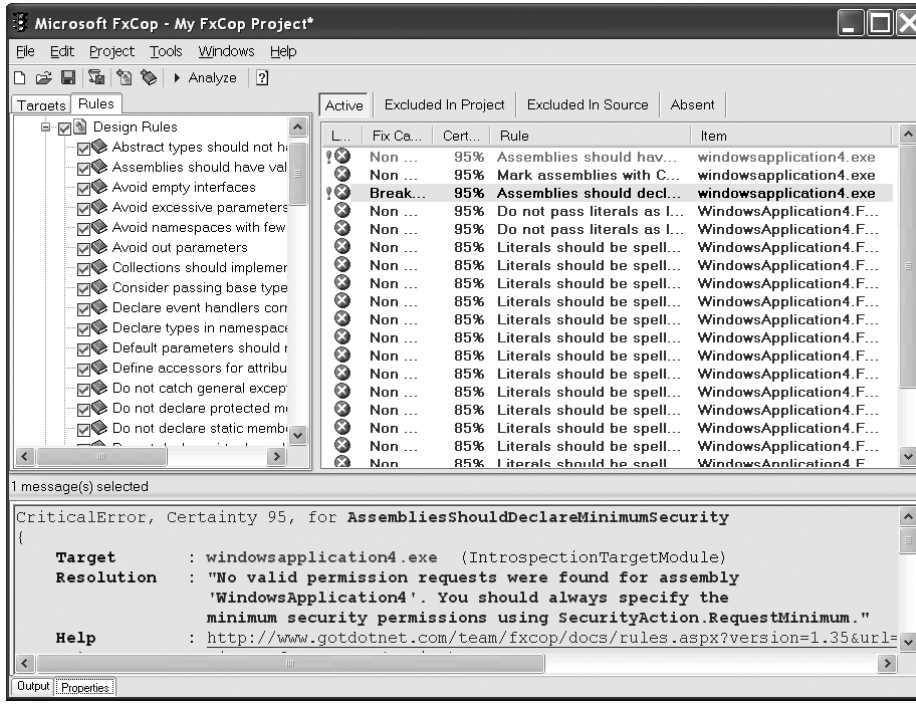
Jak jsem řekl už na začátku, je tento „bleskový kurs" přístupové bezpečnosti CAS jen stručným vysvětlením jeho činnosti, ale měli bychom z něj získat alespoň základní informace pro zbytek kapitoly.

Nástroj FxCop: povinná výbava

Než se pustíme do výkladu otázek bezpečného programování a do nejlepších doporučených postupů, dovolím si vás upozornit na jeden užitečný nástroj, který se jmenuje FxCop a je k dispozici na adrese *http://www.gotdotnet.com*.[32] Uvedený nástroj slouží pro analýzu kódu a kontroluje, jestli dané sestavení (assembly) .NET odpovídá zásadám .NET Framework Design Guidelines, uvedených na webové adrese *http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp*. Nejlépe je proto spustit jej nad každým vytvořeným sestavením a poté nalezené chyby v kódu opravit. Podobně jako u každého jiného nástroje i zde platí, že i když FxCop nenalezne žádná bezpečnostní zranitelná místa, neznamená to, že je kód úplně bezpečný; definuje ovšem alespoň jistou minimální základnu. Výsledky činnosti nástroje nad zkušebním sestavením jsou znázorněny na obrázku 18.5.

Poznámka: Nástroj FxCop umí generovat také soubor formátu XML, do něhož zapíše veškeré případy porušení pravidel. Chcete-li z něj dostat čitelnější zprávu, doplňte za první řádek, `<?xml version="1.0"?>` ještě druhý řádek se značkou `<?xml-stylesheet href="C:\Program Files\Microsoft FxCop\Xml\violationsreport.xsl" type="text/xsl"?>`.

32 Poznámka českého vydavatele: Tyto stránky měly být ale společností Microsoft do měsíce července 2007 uzavřeny. Sledujte aktuální informace.



Obrázek 18.5 Na těchto výsledcích nástroje FxCop vidíme odchylky od zásad .NET Framework Design Guidelines

Dvěma nejběžnějšími chybami, které nástroj FxCop v kódu označuje, je chybějící silný název sestavení a případ, kdy sestavení nespecifikuje požadavky na oprávnění. Podívejme se nyní na obě chyby podrobněji.

Sestavení musí mít silné názvy

Názvy jsou slabou formou autentizace i legitimace („důkazů“, evidence). Pokud vám někdo úplně neznámý předá vypálené CD, na kterém je nějaký soubor s názvem *Excel.exe*, budete mu slepě důvěřovat a spustíte jej? Pokud byste zrovna nutně potřebovali nějaký tabulkový procesor, možná byste po něm sáhli, protože byste si mysleli, že to opravdu Microsoft Excel je. Jak to ale můžete opravdu *vědět*? Systém .NET řeší uvedený problém falšování kódu pomocí silných názvů, které se skládají z prostého textového názvu souboru, čísla verze a informace o jazykové verzi (culture) – a k nim je navíc doplněn veřejný klíč a digitální podpis.

Silný název vytvoříme pomocí nástroje *sn.exe*, který vygeneruje dvojici klíčů silného názvu; pro vytvoření dvojice klíčů zadáme příkaz `sn -k keypair.snk`. Výsledný soubor obsahuje soukromý a veřejný klíč, jenž slouží k podepsání a následnému ověření podepsaných sestavení. (Do výkladu asymetrického šifrování se zde pouštět nebudeme; podrobnější informace najdete ve vhodné literatuře a částečně také v kapitole 5 této knížky.) Pokud je dvojice klíčů určena k „ostrému“ využití, nikoli pouze k experimentál-

ním účelům, musíte ji chránit stejně jako každou jinou dvojici soukromého a veřejného klíče.

Poznamenejme ještě, že silný název je založen na samotné dvojici klíčů, nikoli na certifikátu jako u technologie Authenticode. Jako vývojář můžete vytvořit dvojici klíčů, která definuje váš vlastní, soukromý obor názvů; ostatní nemohou stejný obor názvů používat, protože neznají soukromý klíč. K této dvojici klíčů si můžete následně vyžádat certifikát (pokud chcete), ale u identity silného názvu se certifikáty nepoužívají. To znamená, že podpis kódu nelze identifikovat jménem vydavatele, i když třeba víte, že tento stejný vydavatel kontroluje všechny silné názvy od jisté dvojice klíčů – samozřejmě za předpokladu, že je soukromý klíč zachován v bezpečí a soukromí.

Kromě silných názvů je vhodné doplnit sestavení (assembly) o podpis podle Authenticode a identifikovat tak jeho vydavatele. K tomu musíte sestavení nejprve podepsat silným názvem a poté vytvořit podpis Authenticode nad celým výsledkem. Podpis Authenticode nemůžete generovat jako první, protože následný podpis silného názvu by byl při kontrole podpisu Authenticode považován za pozměnění dat (tampering).

Důležité: Soukromé klíče silných názvů nelze na rozdíl od certifikátů odvolat, a proto musíte přijmout vhodná opatření pro ochranu klíčů. Jednoho velmi důvěryhodného jednotlivce můžete například prohlásit za „mistra klíčů", který bude soukromé klíče uchovávat na disketě, někde v trezoru.

Poznámka: V současné době se u silných názvů používají 1024bitové klíče RSA.

Dále příkazem `sn -p keypair.snk public.snk` extrahujeme z dvojice klíčů příslušný veřejný klíč. Smysl tohoto kroku pochopíte za chvíli. Proces podepisování může proběhnout jen během kompilace kódu a vytvoření binárního souboru, přičemž na informace o klíči se odvoláme pomocí direktivy `[assembly: AssemblyKeyFile(název-souboru)]`. U výchozí aplikace ve Visual Studiu .NET se tato direktiva nachází v souboru *AssemblyInfo.cs* nebo *AssemblyInfo.vb* a v aplikaci Visual Basic .NET vypadá následovně:

```
Imports System.Reflection
<Assembly: AssemblyKeyFileAttribute("c:\keys\keypair.snk")>
```

Uvědomte si ale také, že při takovéto operaci může být soukromý klíč zranitelný vůči prozrazení informací ze strany zlomyslného vývojáře. Toto riziko je možné potlačit pomocí odloženého podepsání, při kterém se používá jen veřejný klíč, nikoli celá dvojice soukromého a veřejného klíče. Vývojáři nemají tím pádem přístup k soukromému klíči a proces úplného podepsání se před vlastní dodávkou kódu zákazníkovi provede příkazem `sn -R assemblyname.dll keypair.snk`. Na vývojových počítačích musíme ale příkazem `sn -Vr assemblyname.dll` obejít ověření neboli verifikaci podpisu, protože takto vytvořené sestavení nemá silný název.

Důležité: Mějte na paměti, že sestavení se silným názvem se může odkazovat jen na jiná sestavení, která také mají silný název.

Odložené podepisování můžeme v sestavení pod Visual Basicem .NET zajistit pomocí následujícího řádku:

```
<Assembly: AssemblyDelaySignAttribute(true)>
```

V jazyce C# vypadá podobná direktiva takto:

```
[assembly: AssemblyDelaySign(true)]
```

Všimněte si tedy, že v jazyce C# je možné výraz *Attribute* z názvu parametru vypustit.

Tip: Vývojáři, kteří na příslušném sestavení pracují denně, by měli vždy provádět odložené podepisování s veřejným klíčem.

Silné názvy sestavení a ASP.NET

Sestavení se silnými názvy, které implementují aplikační logiku webových aplikací, musíme pomocí nástroje .NET Configuration (*Mscorcfg.msc*) nebo *gacutil.exe* uložit do globální mezipaměti cache se sestaveními (global assembly cache, GAC), a to z důvodu, jakým ASP.NET zavádí podepsaný kód.

Podívejme se nyní blíže na oprávnění a na doporučené postupy ohledně požadavků oprávnění.

Stanovení požadavků na oprávnění v sestavení

V požadavcích na oprávnění sdělujeme společné běhové knihovně .NET, jaké operace budeme v kódu potřebovat provádět. I když zaslání požadavku na oprávnění je nepovinné a není pro kompilaci kódu nutnou podmínkou, je tento postup velice vhodný z jistých důvodů, které poznáme až při provádění kódu. Jestliže si kód vyžádá oprávnění pomocí metody `Demand`, ověří běhová knihovna CLR potřebná oprávnění i pro veškerý kód, ze kterého bude náš kód volán. Bez těchto oprávnění požadavek selže. Při tomto ověřování oprávnění se provádí takzvaný *průchod zásobníku* (stack walk). Z pohledu použitelnosti je důležité, aby programový kód dostal veškerá oprávnění, která skutečně potřebuje ke své práci, a z pohledu bezpečnosti je důležité, aby nedostal žádná oprávnění navíc.

Co je to průchod zásobníku?

Průchod zásobníku je důležitou součástí systému zabezpečení v běhové knihovně .NET. Před povolením přístupu k chráněnému prostředku tak běhové prostředí ověří, že potřebné oprávnění pro přístup k prostředku mají i všechny funkce volajícího kódu. Hovoříme o *průchodu zásobníku volání*.

Žádejte minimální množinu oprávnění

Jestliže si vyžádáme správnou množinu oprávnění, bude mít programový kód větší šanci po svém spuštění správně pracovat. Pokud ale jasně nestanovíme minimální množinu oprávnění, kterou program potřebuje ke své činnosti, musíme do aplikace doplnit kód pro ošetření chyb a v něm elegantně obsloužit situace, kdy za běhu některé oprávnění

uděleno nebude. Vhodně vyžádaná oprávnění nám zajistí, že kód dostane skutečně ta oprávnění, která potřebuje – nesmíme tedy žádat širší oprávnění.

Pokud kód nepřistupuje k žádným chráněným prostředkům ani neprovádí žádné bezpečnostně citlivé operace, nemusíme z něj žádná oprávnění požadovat. Pokud například aplikace potřebuje oprávnění `FileIOPermission` jen pro přečtení jediného souboru, stačí do kódu zapsat následující řádek:

```
[assembly: FileIOPermission(SecurityAction.RequestMinimum, Read = @"c:\
files\inventory.xml")]
```

Poznámka: Všechny parametry tohoto deklarativního oprávnění musí být známy již v době kompilace.

Pomocí direktivy `RequireMinimum` nadefinujte minimální možnou množinu oprávnění. Pokud běhová knihovna nedokáže aplikaci přidělit tato minimální oprávnění, vyvolá výjimku `PolicyException` a aplikace se vůbec nespustí.

Nepotřebná oprávnění odmítněte

V zájmu principů nejmenších možných oprávnění je vhodné odmítnout oprávnění, která nepotřebujeme, i když nám je třeba běhová knihovna udělí. Pokud například daná aplikace nebude nikdy provádět souborové operace nebo nebude přistupovat k proměnným systémového prostředí, můžeme do kódu zapsat direktivy:

```
[assembly: FileIOPermission(SecurityAction.RequestRefuse, Unrestricted =
true)]
[assembly: EnvironmentPermission(SecurityAction.RequestRefuse,
Unrestricted = true)]
```

Ocitne-li se takováto aplikace v podezření z útoku vedeného přes soubory, pro který je ale potřeba mít práva přístupu k souborům, máme v rukou jasný důkaz, že náš kód být viníkem nemůže – veškerý přístup k souborům jsme v něm odmítli.

Vyžádejte si volitelná oprávnění

Systém zabezpečení běhové knihovny CLR dává programovému kódu možnost vyžádat si oprávnění navíc, která třeba využije, ale která pro svou činnost nepotřebuje úplně nezbytně. Rozhodnete-li se pro tento typ požadavků, nezapomeňte v kódu zachytit veškeré výjimky, k jejichž vyvolání může dojít při neúspěšném pokusu o udělení práv. Příkladem může být internetová hra, v níž si uživatel může rozehraný stav uložit do lokálního souborového systému. K tomu ale aplikace potřebuje oprávnění `FileIOPermission`; pokud jí je systém neudělí, bude hra funkční, pouze v ní nebude fungovat ukládání rozehraných her. Toto oprávnění si vyžádáme pomocí následujícího zápisu:

```
[assembly: FileIOPermission(SecurityAction.RequestOptional, Unrestricted
= true)]
```

Pokud si v kódu nevyžádáte žádná volitelná oprávnění, budete v něm mít k dispozici všechna oprávnění přidělená díky platným zásadám, minus oprávnění, která jste v aplikaci výslovně odmítli. Zápisem této konstrukce se můžete volitelných oprávnění úplně vzdát:

```
[assembly: PermissionSet(SecurityAction.RequestOptional, Unrestricted =
false)]
```

Sestavení bude mít po této direktivě za běhu přidělenu následující množinu oprávnění:

$$(\text{Opr}_{\text{maximální}} \cap (\text{Opr}_{\text{minimální}} \cup \text{Opr}_{\text{volitelná}})) - \text{Opr}_{\text{odmítnutá}}$$

To znamená, že kód bude mít přidělena ta minimální a volitelná oprávnění, která jsou zapsána v seznamu maximálních oprávnění, minus případná odmítnutá oprávnění.

Imperativní a deklarativní oprávnění

V příkladech kódu jste si jistě všimli, že oprávnění definovaná na úrovni sestavení jsou v jazyce C#, respektive Visual Basic .NET, zapsána do hranatých, respektive úhlových závorek. Tyto zápisy definují takzvaná *deklarativní oprávnění*. Druhou možností je zavést imperativní bezpečnost, tedy vytvořit v kódu objekty oprávnění. Příkladem může být příkaz `new FileIOPermission(FileIOPermissionAccess.Read = @"c:\files\inventory.xml").Demand();` – ten se pokusí získat oprávnění k přečtení souboru XML a v případě neúspěchu vyvolá výjimku.

Nezapomeňte proto v kódu odchytit všechny takovéto výjimky – jinak se provádění aplikace zastaví.

Obě metody mají svoje výhody a nevýhody. Deklarativní oprávnění se v kódu snadno zapisují i hledají; můžeme si je prohlížet pomocí nástroje Permissions View (*permview*), který nám pomůže i při auditech a revizích kódu (pro zobrazení deklarací zde napíšeme přepínač */decl*). Ani při změnách v toku řízení programu tyto kontroly neobejdeme a oprávnění můžeme navíc aplikovat na celé třídy.

Největší nevýhodou deklarativních oprávnění je to, že je musíme znát předem, již v době kompilace.

Další informace: Požadavky daného sestavení na oprávnění můžeme zjistit příkazem **caspol -a -resolveperm myassembly.exe**, který vypíše, jaká oprávnění by po zavedení daného sestavení byla přidělena, nebo pomocí nástroje **permview** z balíku .NET Framework sdl, který zobrazuje požadavky daného sestavení, tedy vstup do vyhodnocení zásad, kde oprávnění může, ale nemusí být přiděleno.

Příliš horlivá volání Assert

Běhová knihovna .NET CLR nabízí metodu s názvem `Assert`, pomocí které může kód aplikace – a jeho prostřednictvím i další volající – „uplatnit oprávnění“ (assert permissions), tedy provádět operace, k nimž má tento kód sám oprávnění, ale jeho volající je již mít nemusí. Jinými slovy, volání `Assert` znamená: „Vím, co dělám: důvěřuj mi.“ Po něm v kódu následuje určitá neškodná operace, ke které by normálně musel volající mít potřebné oprávnění.

Důležité: Nepleťte si bezpečnostní metodu `CodeAccessPermission.Assert` ze společné běhové knihovny .NET s funkcí `assert` klasického C nebo C++, případně s metodou `Debug.Assert` systému .NET Framework. Zmíněné metody provádějí vyhodnocení jistého výrazu, a pokud je jeho výsledek roven *false*, zobrazí diagnostickou zprávu.

Dejme tomu, že například aplikace čte určitý konfigurační nebo vyhledávací soubor, ale volající kód nemá oprávnění k žádným souborovým I/O operacím. Pokud spolehlivě víte, že jsou operace vašeho kódu s tímto souborem neškodné, můžete metodou `Assert` „uplatnit", že budete se souborem pracovat bezpečně.

K tomu je ovšem třeba říci, že toto uplatnění může být v některých situacích bezpečné a v některých ne. Volání `Assert` se zpravidla používá v situaci, kdy vysoce důvěryhodnou knihovnu používá nějaký méně důvěryhodný kód a kdy je nutné zastavit průchod zásobníku. Představte si například, že vytvoříte implementaci třídy pro přístup k souborům přes rozhraní USB (Universal Serial Bus); tato třída bude mít název *UsbFileStream* a bude odvozena od třídy `FileStream`. Nový kód bude k souborům přistupovat prostřednictvím volání API rozhraní USB Win32, ale rozhodně nebude od všech svých volajících vyžadovat oprávnění k volání neřízeného kódu; bude mu stačit oprávnění `FileIOPermission`. Kód třídy `UsbFileStream` provede proto ve volání `Assert` operaci `UnmanagedCode` (aby mohl pracovat s Win32 API) a vyžádá si oprávnění `FileIOPermission`, se kterým potvrdí, že volající má povoleno provést danou I/O operaci nad souborem.

Jestliže ale nyní nějaký kód převezme název souboru z nedůvěryhodného zdroje, například od uživatele, a poté jej otevře pro operaci zkrácení, jistě si nepočíná bezpečně. Co když třeba uživatel předá do programu požadavek, jako například *../../boot.ini*? Vymaže program skutečně soubor *boot.ini*? Je možné, že se to opravdu stane – zejména pokud je přístupový seznam (ACL) na tomto souboru slabý, pokud aplikace běží pod účtem administrátora nebo pokud je soubor umístěn na diskovém oddíle FAT.

Při provádění bezpečnostní revize kódu se podívejte na veškerá bezpečnostní „uplatnění" `Assert` a raději dvakrát zkontrolujte, jestli jsou následné operace opravdu neškodné, zejména pokud v kódu objevíte samostatné `Assert` bez odpovídajícího `Demand` nebo pokud u slabého oprávnění najdete `Assert` a `Demand` zároveň. Takto můžeme například přes `Assert` zavolat neřízený kód a poté si přes `Demand` vyžádat oprávnění pro přístup k proměnné systémového prostředí.

Poznámka: Jestliže v kódu potřebujeme uplatnit nějaké oprávnění ve volání `Assert`, musíme toto oprávnění nejprve mít přidělené.

Důležité: Dávejte si velký pozor zejména na situace, kdy v kódu pomocí zápisu *SecurityPermissionFlag.UnmanagedCode* uplatňujete oprávnění k volání neřízeného kódu; pokud se vám v kódu podaří udělat chybu, můžete nakonec způsobit neúmyslné vyvolání neřízeného kódu.

Další informace k voláním Demand a Assert

Při vytváření aplikací s metodami `Demand` a `Assert` je třeba dodržovat několik jednoduchých pravidel. Z kódu je tak především nutné metodou `Assert` uplatnit jedno nebo více oprávnění v případě, kdy provádíme nějakou privilegovanou, i když bezpečnou operaci a kdy zároveň nechceme toto oprávnění požadovat po volajícím. Poznamenejme, že kód musí mít oprávnění, které v metodě `Assert` uplatňujeme, a také oprávnění *SecurityPermissionFlag.Assert*, tedy oprávnění k volání metody `Assert`.

Pokud například uplatníme oprávnění `FileIOPermission`, musí mít náš kód toto oprávnění uděleno, ale volající kód je již nepotřebuje. Jestliže uplatníme oprávnění `FileIOPermission` za situace, kdy je v kódu uděleno nemáme, provede se průchod zásobníku a poté dojde k vyvolání výjimky.

Jak jsem se již zmínil, pokud vyžadujeme vlastnictví příslušného oprávnění od volajících, musíme si je *vyžádat* („poptávat“) z kódu v požadavku, který odešleme metodou `Demand`. Dejme tomu, že například aplikace odesílá ostatním jistá oznámení pomocí elektronické pošty a že za tímto účelem definuje jisté vlastní oprávnění s názvem `EmailAlertPermission`. Jakmile nějaký volající kód vyvolá naši aplikaci, budeme pomocí metody `Demand` vyžadovat toto oprávnění u všech volajících; pokud volající nemá oprávnění `EmailAlertPermission` přiděleno, požadavek selže.

Důležité: Při požadavku (`Demand`, „poptávce“) se udělení příslušného oprávnění nekontroluje u kódu, který provedl metodu `Demand`, ale jen u jeho volajících. Máme-li například ve funkci `Main` přidělena omezená oprávnění, proběhne u ní kontrola jakéhokoli požadavku úspěšně – protože tato funkce již nemá žádného volajícího. Proto potřebujete-li ověřit oprávnění vlastního kódu, musíte v něm buďto vyvolat určitou funkci a metodu `Demand` vyvolat teprve zde – tím se odhalí oprávnění volajícího – anebo pomocí metody `SecurityManager.IsGranted` přímo zjistit oprávnění přidělená danému sestavení (a to pouze tomuto sestavení, protože volající již stejná oprávnění mít nemusí). To ale rozhodně neznamená, že do funkce `Main` můžete napsat jakýkoli škodlivý kód a že bude fungovat! Pokud kód vyvolává třídy, které se pokoušejí o provedení potenciálně nebezpečných operací, proběhne následně průchod zásobníku a kontrola oprávnění.

Důležité: Jestliže z kódu aplikace vyvoláváte jiný kód, který požaduje určitá oprávnění, neprovádějte stejný požadavek z vlastní aplikace. Důvod souvisí s rychlostí zpracování – nemá smysl provádět takto dvojí průchod zásobníku. Jinými slovy, pokud vyvoláte někde z kódu metodu `Environment.GetEnvironmentVariable`, nemusíte již zvlášť požadovat oprávnění `EnvironmentPermission`, protože systém .NET Framework to udělá za nás.

Napsat kód, který uplatňuje (`Assert`) a požaduje (`Demand`) oprávnění, není nijak těžké. Budeme-li například pokračovat ve výše uvedeném scénáři s e-mailovými upozorněními, může kód, který již přímo komunikuje s podsystémem elektronické pošty, požadovat u všech volajících oprávnění `EmailAlertPermission` (to je námi definované, vlastní oprávnění). Poté tento kód zapíše e-mailovou zprávu do portu SMTP a přitom může uplatnit oprávnění `SocketPermission`. V tomto scénáři mohou volající programy využívat náš kód k zasílání elektronické pošty, ale již nebudou mít možnost odesílat data do

libovolného portu – přestože to umožňuje oprávnění `SocketPermission` (to je uděleno jen našemu kódu, nikoli volajícímu).

Kde je oprávnění UnmanagedCode?

Možnost volání neřízeného kódu je vysoce privilegovaným oprávněním. Jakmile totiž „unikneme" z prostoru řízeného kódu, můžeme na počítači dělat v podstatě cokoliv, samozřejmě podle možností daného uživatelského účtu. Kde je tedy oprávnění `UnmanagedCode`? Je „zastrčené" dovnitř jiného oprávnění.

U některých funkcí provádíme jednoduché „binární" rozhodnutí ano/ne, zatímco jiné jsou složitější. Také možnost volání neřízeného kódu je binárního typu – tento kód zkrátka volat můžeme, nebo nemůžeme. Možnost přístupu k souborům, kterou ovládá třída oprávnění `FileIOPermission`, je oproti tomu složitější: kód může mít přiděleno právo čtení z jednoho souboru a zápisu do jiného souboru – nejedná se tedy o jednoduché „binární" rozhodování. Oprávnění k volání neřízeného kódu je definováno různými příznaky ve třídě `SecurityPermission`, jak vidíme v následujícím řádku:

```
[SecurityPermission(SecurityAction.Assert, UnmanagedCode=true)]
```

A nakonec, metodu `Permission.Assert` nemůžeme volat dvakrát po sobě – jinak vyvolá výjimku. Proto potřebujete-li v aplikaci uplatnit více než jedno oprávnění, musíte nejprve vytvořit množinu oprávnění, vložit do ní jednotlivá oprávnění a nakonec metodou `Assert` uplatnit celou množinu, například:

```
try {
    PermissionSet ps =
        new PermissionSet(PermissionState.None);
    ps.AddPermission(new FileDialogPermission
        (FileDialogPermissionAccess.Open));
    ps.AddPermission(new FileIOPermission
        (FileIOPermissionAccess.Read.@"c:\files"));
    ps.Assert();
} catch (SecurityException e) {
    // auuu!
}
```

Asertivní okno při uplatnění musí být malé

Po dokončení operace, ke které potřebujeme oprávnění speciálně uplatněné metodou `Assert`, je třeba toto uplatnění odvolat pomocí metody `CodeAccessPermission.RevertAssert`. To je opět příklad s nejmenším možným oprávněním: uplatněné a přidělené oprávnění použijeme jen po dobu, kdy je to nezbytně nutné, a poté je „poslušně" vrátíme.

Následující ukázka programového kódu v C# používá vhodnou kombinaci uplatnění (`Assert`), požadavků či „poptávek" (`Demand`) a vracení (`Revert`) oprávnění pro rozesílání e-mailových upozornění. Volající musí mít oprávnění k odesílání elektronické pošty, a pokud je skutečně má, může odesílat e-mail přes soket SMTP, přestože oprávnění k otevření obecného soketu již nemá:

```
using System;
using System.Net;
using System.Security;
using System.Security.Permissions;

// Toto je pouze fragment kódu; chybí zde třídy a obory názvů.

static void SendAlert(string alert) {
    // Požadavek po oprávnění volajícího k odesílání e-mailu.
    new EmailAlertPermission(
        EmailAlertPermission.Send).Demand();

    // Zde v kódu otevřeme konkrétní port na konkrétním serveru SMTP.
    NetworkAccess na = NetworkAccess.Connect;
    TransportType type = TransportType.Tcp;
    string host = "mail.northwindtraders.com";
    int port = 25;
    new SocketPermission(na, type, host, port).Assert();

    try {
        SendAlertTo(host, port, alert);
    } finally {
        // Vždy oprávnění vrátíme, i při selhání
        CodeAccessPermission.RevertAssert();
    }
}
```

Pokud se volání metod `Assert`, `Deny` a `PermitOnly` nacházejí na stejném objektu, provádí se nejprve akce `Deny`, poté `Assert` a nakonec `PermitOnly`.

Představte si metodu `A()`, která zavolá metodu `B()`, z ní se potom vyvolá metoda `C()`, přičemž metoda `A()` odepře oprávnění `ReflectionPermission`. Metoda `C()` může ale ještě oprávnění `ReflectionPermission` uplatnit (voláním `Assert`), samozřejmě za podmínky, že sestavení, které ji obsahuje, má toto oprávnění přiděleno. Proč to? Protože jakmile běhová knihovna narazí na uplatnění (metodu `Assert`), zastaví průchod zásobníku a k odepřenému oprávnění v metodě `A()` se již nedostane. Následující příklad kódu ukazuje celý tento postup na jediném sestavení:

```
private string filename = @"c:\files\fred.txt";

private void A() {
    new FileIOPermission(
        FileIOPermissionAccess.AllAccess,filename).Deny();
    B();
}

private void B() {
    C();
}

private void C() {
    try {
        new FileIOPermission(
            FileIOPermissionAccess.AllAccess,filename).Assert();
        try {
            StreamWriter sw = new StreamWriter(filename);
            sw.Write("Ahoj!");
            sw.Close();
```

```
            } catch (IOException e) {
                Console.Write(e.ToString());
            }
        } finally {
            CodeAccessPermission.RevertAssert();
        }
    }
```

Jestliže volání `Assert` z metody `C()` odstraníte, vyvolá kód při instanciaci třídy `StreamWriter` výjimku `SecurityException`, protože kódu je již oprávnění odepřeno.

Požadavky a požadavky na odkaz

Už jsme si ukázali příklad kódu, který ke své správné činnosti „poptává" neboli *požaduje* (demand) jistá oprávnění. Většina tříd v systému .NET Framework má již určité požadavky přidružené, takže při volání třídy, která přistupuje k nějakému chráněnému prostředku, již není nutné zasílat další požadavek. Třída `System.IO.File` například při otevření jakéhokoli souboru z kódu automaticky požaduje oprávnění `FileIOPermission`. Pokud někde v kódu používáte třídu `File`, ale ještě vyvoláte zvláštní požadavek po oprávnění `FileIOPermission`, znamená to jen redundantní a zbytečný průchod zásobníku. Pomocí požadavku musíte ochránit zejména svoje vlastní prostředky, ke kterým jsou potřeba vlastní oprávnění.

U požadavku na odkaz (link demand) se provádí bezpečnostní kontrola až během běhové kompilace volající metody (to je kompilace v režimu „právě včas", just-in-time, JIT) a kontroluje se pouze bezprostředně vyšší volající kód, ze kterého byl náš kód vyvolán. Pokud tento volající nemá dostatečná oprávnění k odkazu na náš kód – tedy pokud například z našeho kódu od volajícího kódu požadujeme za běhu oprávnění `IsolatedStorageFilePermission` – není odkaz povolen a běhová knihovna vyvolá při zavedení a spuštění kódu výjimku.

Při požadavcích na odkaz se neprovádí úplný průchod zásobníku, a proto je náš kód opět zranitelný vůči útoku s vylákáním (luring attack) – to znamená, že méně důvěryhodný kód zavolá náš vysoce důvěryhodný kód a zneužije jej k provedení neoprávněných operací. Požadavek na odkaz vyjadřuje pouze ta oprávnění, která musí mít volající kód při odkazu na náš kód; nevyjadřuje již, jaká oprávnění musí mít volající kód při vlastním spuštění kódu. Tato oprávnění je možné zjistit pouze průchodem zásobníku.

Příklad bezpečnostní chyby s voláním LinkDemand

Konečně se dostáváme ke slíbenému problému. Podívejte se na následující ukázku kódu:

```
[PasswordPermission(SecurityAction.LinkDemand, Unrestricted=true)] [Regis
tryPermissionAttribute(SecurityAction.PermitOnly,
    Read=@"HKEY_LOCAL_MACHINE\SOFTWARE\AccountingApplication")]

public string returnPassword() {
    return (string)Registry
        .LocalMachine
        .OpenSubKey(@"SOFTWARE\AccountingApplication\")
        .GetValue("Password");
```

```
}

...

public string returnPasswordWrapper() {
    return returnPassword();
}
```

Ano, vím a jistě víte i vy – tento kód je nebezpečný proto, že přenáší tajné informace dovnitř samotného kódu, ale budu hovořit ještě o něčem jiném. Při volání funkce `returnPassword` musí mít volající kód vlastní oprávnění s názvem `PasswordPermission`. Pokud by kód provedl volání `returnPassword` bez uvedeného oprávnění, vyvolala by běhová knihovna bezpečnostní výjimku a kód by se k heslu nedostal. Jestliže ale kód vyvolá obálkovou funkci `returnPasswordWrapper`, odešle se požadavek na odkaz pouze nad vyvolanou funkcí `returnPassword` a nikoli nad kódem, který vyvolal funkci `returnPasswordWrapper`, protože požadavek na odkaz jde vždy pouze o jednu úroveň do hloubky. Kód, který zavolal funkci `returnPasswordWrapper`, se tím pádem k heslu dostane.

Protože požadavky na odkaz se provádějí až za běhu, v režimu JIT a protože se při nich ověřuje jen tolik, jestli má přímý volající potřebné oprávnění, jsou rychlejší než úplné požadavky, ale zároveň poskytují potenciálně slabší bezpečnostní mechanismus.

Jaké je z tohoto „příběhu" ponaučení? Nikdy nepoužívejte požadavky na odkaz, pokud jste kód nepodrobili opravdu důkladné revizi. Provedení jednoho požadavku i s úplným průchodem zásobníku trvá sotva několik mikrosekund, takže pokud nahradíte plnohodnotné požadavky za požadavky na odkaz, rozdíl v rychlosti zpracování ani nepostřehnete. Máte-li ale požadavky na odkaz v kódu, raději si dvakrát zkontrolujte, jestli v nich nejsou nějaké bezpečnostní chyby – a to zejména tehdy, pokud nemůžete splnění příslušných kontrol v době odkazování zaručit úplně u každého volajícího. Podobně pokud vyvoláte z aplikace nějaký cizí kód, který provádí požadavky na odkaz, zeptejte se: neprovádíte ve svém kódu nějaké operace, které by mohly tyto požadavky narušit? A nakonec, jestliže je požadavek na odkaz definován nad virtuálním odvozeným elementem, zkontrolujte, jestli stejný požadavek existuje také na příslušném bázovém elementu.

Důležité: Reflexní vrstva běhové knihovny provádí při operaci požadavku `Demand` úplný průchod zásobníku, přičemž u všech případů pozdní vazby (late-bound) pracuje pod stejnými oprávněními; účelem je zabránit zneužití požadavku `LinkDemand` i zneužití reflexe (to je proces získávání informací o sestaveních a typech a dále vytvoření, vyvolání a přístupu k instancím typu za běhu). Tím je potlačeno riziko možného přístupu k chráněnému členu prostřednictvím reflexe, ke kterému by jinak mohlo dojít i v případě, že by při normální včasné vazbě (early-bound) nebyl přístup dovolen. Při úplném průchodu zásobníku se ovšem mění sémantika požadavku na odkaz, vyvolaného původně prostřednictvím reflexe, a navíc je tato operace pomalejší; proto je vhodnější použít raději úplný požadavek. Tím je celý proces nejen rychlejší, ale i srozumitelnější.

S atributem SuppressUnmanagedCodeSecurityAttribute opatrně

Jestliže v kódu pracujete s atributem `SuppressUnmanagedCodeSecurityAttribute`, dávejte opravdu hodně velký pozor. Normálně může volání do neřízeného kódu proběhnout úspěšně jen tehdy, pokud mají všechny volající moduly oprávnění volat neřízený kód. Pokud na metodu, která provádí volání do neřízeného kódu, aplikujeme vlastní atribut `SuppressUnmanagedCodeSecurityAttribute`, „poptávka" neboli požadavek se tím potlačí. Namísto úplného požadavku provede kód pouze požadavek na odkaz s možností volání neřízeného kódu. Jestliže z kódu budete volat větší množství nativních funkcí Win32, může se tím kód výrazně urychlit, ale řešení je zároveň i nebezpečné. Následující fragment kódu aplikuje atribut `SuppressUnmanagedCodeSecurityAttribute` na metodu (funkci) `MyWin32Function`:

```
using System.Security;
using System.Runtime.InteropServices;
...
public class MyClass {
    ...
    [SuppressUnmanagedCodeSecurityAttribute()]
    [DllImport("MyDLL.DLL")]
    private static extern int MyWin32Function(int i);

    public int DoWork() {
        return MyWin32FunctionC0x42);
    }
}
```

Ve všech metodách, které uvedený atribut používají, raději dvakrát zkontrolujte bezpečnost kódu.

Důležité: Volání `LinkDemand` a atribut `SuppressUnmanagedCodeSecurityAttribute` mají něco společného – u obou se musíme rozhodnout mezi rychlostí kódu a jeho bezpečností. Nezapínejte proto uvedené prvky nahodile a vždy si pečlivě prověřte, jestli potenciální urychlení kódu stojí za riziko vyšší bezpečnostní zranitelnosti. Uvedené funkce nezapínejte, dokud měřením nepotvrdíte urychlení kódu (pokud zde vůbec nějaké urychlení je). Pokud se rozhodnete zapnout funkci `SuppressUnmanagedCodeSecurity`, dodržujte tyto doporučené postupy: nativní metody deklarujte jako `private` nebo `internal` a všechny argumenty ve volání metody je nutné validovat.

Vzdálené požadavky

Pokud je určitý objekt možné volat vzdáleně (tedy pokud je objekt odvozen od třídy `MarshalByRefObject`) a pokud k němu přistupujeme na dálku přes hranice procesů nebo počítačů, bezpečnostní kontroly přístupového kódu jako `Demand`, `LinkDemand` a `InheritanceDemand` se neprovádějí. To například znamená, že bezpečnostní kontroly neprocházejí v prostředí webových služeb přes protokol SOAP. Bezpečnostní kontroly přístupového kódu procházejí ovšem přes aplikační domény. Stojí také za zmínku, že vzdálené volání je podporováno jen v plně důvěryhodných prostředích. Jinými slovy,

pokud je nějaký kód považován na klientu za plně důvěryhodný, nemusí být plně důvěryhodným i v kontextu serveru.

Omezte přístup k vašemu kódu

Volání některých metod našeho vlastního kódu je někdy nevhodné dovolit úplně jakémukoli, i nedůvěryhodnému kódu. Metoda může například poskytovat jisté omezené informace nebo třeba z různých důvodů provádí jen minimální kontroly chyb. Řízený kód nabízí několik způsobů pro omezení přístupu k metodám; nejjednodušší je omezit obor třídy (scope), sestavení nebo odvozené třídy. Poznamenejme přitom, že odvozená třída může být méně důvěryhodná než třída, ze které je odvozena; konec konců nikdy nevíme, kdo se rozhodne z naší třídy odvodit nějaký jiný kód. Důvěryhodnost kódu nelze mlčky vyvozovat ani z klíčového slova `protected`, které z bezpečnostního hlediska neznamená vůbec nic. Člen třídy typu `protected` je dostupný zevnitř třídy, v níž byl sám deklarován, a také z jakékoli třídy, která je z této původní třídy odvozená – podobně jako je klíčové slovo `protected` definováno ve třídách C++.

Třídy je dále vhodné zapečetit. *Zapečetěné* třídy (sealed classes) – ve Visual Basicu se označují jako nedědděné, `NotInheritable` – jsou zkrátka takové třídy, které nelze podědit (odvodit). Jinými slovy, zapečetěnou třídu nelze použít jako bázovou třídu pro definici jiné třídy. Pokud takovouto třídu nadefinujeme v kódu, výrazně tím omezíme kód, který bude naše třídy dědit. Nezapomeňte přitom, že žádnému kódu, vytvořenému děděním z naší třídy, nemůžeme důvěřovat – jedná se tedy o rozumnou „hygienu“ objektově orientovaného programování.

Dále můžeme omezit metody přístupu pro volající jen na zvlášť vybrané metody. Podobně v rámci deklarativní bezpečnosti můžeme řídit dědění tříd. Pomocí volání `InheritanceDemand` nařídíme pro odvozené třídy zadanou identitu nebo oprávnění, případně budeme požadovat určitou identitu nebo oprávnění v každé třídě, která přepisuje (override) nějaké metody. Takto například nadefinujeme třídu, kterou je možné vyvolat pouze z takového kódu, jenž má oprávnění `EnvironmentPermission`:

```
[EnvironmentPermission
(SecurityAction.InheritanceDemand, Unrestricted=true)]
public class Karel {
    ...
}

class Pepa : Karel {
    ...
}
```

V tomto příkladu musí třída `Pepa`, která je vytvořena zděděním třídy `Karel`, mít oprávnění `EnvironmentPermission`.

Požadavek po dědění jde ještě o jeden krok dále – můžeme pomocí ní omezit, jaký kód může přepisovat virtuální metody. Takto můžeme například požadovat vlastní oprávnění `PrivateKeyPermission` u každé metody, která se pokusí přepsat virtuální metodu `SetKey`:

```
[PrivateKeyPermission
(SecurityAction.InheritanceDemand, Unrestricted=true)]
public virtual void SetKey(byte [] key) {
```

```
        m_key = key;
        DestroyKey(key);
    }
```

Kromě toho je možné omezit, z jakých sestavení je možné kód vyvolat; povolená sestavení definujeme pomocí jejich silného názvu:

```
[StrongNameIdentityPermission(SecurityAction.LinkDemand, PublicKey="00240
fd981762bd0000...172252f490edf20012b6")]
```

A kód můžeme také svázat zpět se serverem, kde tento kód vznikl. Uvedená funkce je podobná mechanismu šablony SiteLock, který byl popsán v kapitole 16. Celý postup ukazuje následující kód, ale pamatujte si – ani tento kód není náhradou dobrého zabezpečení přístupového kódu. Nepište žádný nebezpečný kód ve falešné naději, že přece bude instanciován jen z jednoho konkrétního webového serveru a že se k němu tedy žádný zlomyslný uživatel nedostane. Pokud nevíte, proč byste to neměli dělat, vzpomeňte si na problémy s křížovými skripty mezi servery!

```
private void function(string[] args) {
    try {
        new SiteIdentityPermission(
            @"*.explorationair.com").Demand();
    } catch (SecurityException e){
        // kód nebyl vyvolán ze serveru Exploration Air
    }
}
```

V kódu XML ani konfiguračních souborech nesmí být citlivá data

Vím, že jsem to už říkal na začátku této kapitoly, ale stojí za to si připomenout ještě jednou. Do konfiguračních souborů, jako je například *web.config*, můžete ukládat jakékoli údaje, jen pokud nejsou citlivé. Hesla, klíče a řetězce pro připojení k databázi je ale třeba uložit někde, kde budou před zraky útočníků skryté. Umístit citlivá data do registru je mnohem bezpečnější, než je takto nechat „napospas ďáblu". Je jasné, že se tímto vzdáváme možnosti nasazení aplikace jediným příkazem `xcopy`, ale život už je takový.

ASP.NET verze 1.1 podporuje volitelné rozhraní Data Protection API pro šifrování tajných dat, uložených v chráněném klíči systémového registru. (Podrobnější informace o rozhraní DPAPI jsou uvedeny v kapitole 9.) Uvedený mechanismus můžeme využít především v konfiguračních sekcích `<processModel>`, `<identity>` a `<sessionState>`. Při zapnutém šifrování obsahuje konfigurační soubor odkaz na klíč registru a na hodnotu, v níž jsou tajné informace uloženy. Chráněné tajné informace vytvoříme v ASP.NET pomocí malé utility příkazového řádku s názvem `aspnet_setreg`. Podívejme se na malý příklad konfiguračního souboru, který načítá uživatelské jméno a heslo, pod nímž se spouští pracovní proces ASP.NET:

```
<system.web>
   <processModel
      enable="true"
      userName="registry:HKLM\Software\SomeKey,userName"
      password="registry:HKLM\Software\SomeKey,passWord"
```

```
        ...
    />

</system.web>
```

Tajné informace zde chrání funkce `CryptProtectData` pomocí šifrovacího klíče definovaného na úrovni počítače. Ta sice nedokáže potlačit veškeré hrozby spojené s ukládáním tajných informací – pokud má někdo k počítači fyzický přístup, dostane se i k uloženým tajným datům – ale oproti ukládání dat do běžného konfiguračního souboru v nešifrované podobě znamená výrazně vyšší bezpečnost.

Uvedená technika neslouží ovšem pro ukládání libovolných aplikačních dat – je určena pouze pro uživatelská jména a hesla, která se používají v identitě procesu ASP.NET, a pro data o spojení stavové služby.

Kontrolujte sestavení, která umožňují částečnou důvěru

Dobře si pamatuji na den, kdy bylo přijato rozhodnutí doplnit do systému .NET atribut `AllowPartiallyTrustedCallersAttribute`. Zdůvodnění bylo naprosto logické: většina útoků pochází přece z Internetu, jehož kódu je možné částečně důvěřovat – to znamená, že kódu povolíme provádění jistých operací a jiné nikoli. Firma může tak například vynucovat bezpečnostní politiku, podle níž bude volání kódu z Internetu do otevřeného soketového spojení zpět na zdrojovém serveru povoleno, ale nad tímto spojením nebude možné tisknout dokumenty ani číst a zapisovat soubory. Přijali jsme tudíž rozhodnutí nedovolit částečně důvěryhodnému kódu přístup k určitým sestavením, která se dodávají s knihovnou CLR a systémem .NET Framework – a to znamená ve výchozím nastavení také veškerý kód vytvořený libovolnou ze třetích stran, tedy i vámi samotnými. Tím se v cílovém prostředí výrazně zúží potenciální plocha útoku. Tento den si opravdu dobře pamatuji, protože zmíněný nový atribut zakazuje nahodilé volání inkriminovaného kódu z potenciálně nepřátelského internetového kódu. Nastavení této volby provede na základě svého rozhodnutí vývojář aplikace.

Jestliže vytváříte kód, který může být vyvolán z částečně důvěryhodného kódu, a jestliže jste jej podrobili důkladnými revizím i bezpečnostnímu testování, můžete zmíněný typ volání povolit pomocí vlastního atributu `AllowPartiallyTrustedCallersAttribute`, definovaného na úrovni sestavení:

```
[assembly:AllowPartiallyTrustedCallers]
```

Sestavení, která povolují spouštění od částečně důvěryhodných volajících, by neměla nikdy dávat k dispozici objekty pocházející z jiných sestavení, která částečně důvěryhodné volající nepřipouštějí.

Důležité: Nezapomeňte, že sestavení bez silného názvu se z částečně důvěryhodného kódu dají vyvolávat vždy.

A nakonec, pokud váš kód není plně důvěryhodný, nemůže často vyvolávat jiný kód, který vyžaduje plně důvěryhodné volající – jako jsou například sestavení se silnými názvy, u nichž atribut `AllowPartiallyTrustedCallersAttribute` definován není.

Dále si dávejte pozor na následující scénář, ve kterém se sestavení rozhodne odmítnout určitá oprávnění:

- Sestavení A se silným názvem nemá definován atribut `AllowPartiallyTrustedCallersAttribute`.
- Sestavení B se silným názvem pomocí požadavku oprávnění odmítne oprávnění, což znamená, že se stane jen částečně důvěryhodným – nemá už totiž plnou důvěru.
- Nyní již sestavení B nemůže vyvolávat kód v sestavení A, protože A částečně důvěryhodné volající nepodporuje.

Důležité: Atribut `AllowPartiallyTrustedCallersAttribute` je vhodné aplikovat jen za podmínky, že kód podrobíte důkladné revizi, že plně posoudíte jeho bezpečnostní důsledky a že přijmete nezbytná opatření pro obranu proti útoku.

Kontrolujte správnost řízených obálek nad neřízeným kódem

Jestliže provádíte volání do neřízeného kódu – a hodně lidí to z důvodu flexibility opravdu dělá – musíte pečlivě zkontrolovat, že je volající kód dobře napsaný a bezpečný. Pokud navíc používáte atribut `SuppressUnmanagedCodeSecurityAttribute`, který umožňuje volání z řízeného kódu do neřízeného bez průchodu zásobníku, položte si otázku, jestli je bezpečné *nevyžadovat* od veřejných („cizích“) volajících oprávnění přístupu k neřízenému kódu.

Problémy s delegáty

Delegáty jsou v principu podobné ukazatelům funkcí z C/C++ a systém .NET Framework pomocí nich podporuje události. Jestliže v kódu přijímáte delegáty, netušíte samozřejmě předem, jak vypadá jejich kód, kdo je jejich autorem a jaké jsou úmysly tohoto autora. Víte jen jediné – který delegát má být při vygenerování události z vašeho kódu vyvolán. Nevíte ani to, jaký kód provádí registraci delegátu. Dejme tomu, že například naše vlastní komponenta `AppA` vyvolává události, zatímco `AppB` registruje delegát pomocí volání ovladače `AddHandler`. Delegátem může být potenciálně jakýkoli kód, například kód, který pozastavuje (blokuje) proces, nebo jej dokonce ukončí voláním `System.Environment.Exit`. Jakmile tudíž aplikace `AppA` sama vyvolá událost, v důsledku její „obsluhy“ skončí nebo provede ještě něco horšího.

Podívejme se, jak je možné uvedený problém oslabit – delegáty jsou silně typované, takže pokud v kódu povolíme jen delegáty s přesně definovanou signaturou („podpisem“) funkce, jako například:

```
public delegate string Function(int count, string name, DateTime dt);
```

pak kód, který se pokusí delegát registrovat a namísto toho registruje volání `System.Environment.Exit`, nakonec selže, protože uvedená metoda má jinou signaturu.

A nakonec můžeme omezit okruh činností povolených v kódu delegátu – k tomu nad požadovanými, respektive odepíranými, oprávněními zavoláme metodu `PermitOnly`, respektive `Deny`. Pokud například budeme chtít, aby měl delegát možnost pouze číst jistou proměnnou systémového prostředí a nic více, můžeme před vyvoláním události provést následující příkaz:

```
new EnvironmentPermission(
    EnvironmentPermissionAccess.Read, "USERNAME").PermitOnly();
```

Nezapomeňte, že se metoda `PermitOnly` vztahuje pouze na kód delegáta (tedy na kód, vyvolaný při vzniku události), nikoli na kód, kterým byl delegát registrován. Na první pohled to nemusí být úplně jasné.

Problémy se serializací

Zvláštní pozornost věnujte také třídám, které implementují rozhraní `ISerializable` a jejichž instance (objekty) mohou obsahovat citlivé informace. Vidíte, jaké zranitelné místo obsahuje následující kód?

```
public void WriteObject(string file) {
    Password p = new Password();
        Stream stream = File.Open(file, FileMode.Create);
    BinaryFormatter bformatter = new BinaryFormatter();
    bformatter.Serialize(stream, p);
    stream.Close();
}

[Serializable()]
public class Password: ISerializable {
    private String sensitiveStuff;
    public Password() {
        sensitiveStuff=GetRandomKey();
    }

    // Deserializace vektoru.
    public Password (SerializationInfo info, StreamingContext context) {
        sensitiveStuff =
            (String)info.GetValue("sensitiveStuff", typeof(string));
    }

    // Serializační funkce.
    public void GetObjectData
        (SerializationInfo info, StreamingContext context) {
        info.AddValue("sensitiveStuff", sensitiveStuff);
    }
}
```

Jak vidíte, útočník nemá sice přímý přístup k tajným datům uloženým v řetězci `sensitiveStuff`, ale může klidně aplikaci přinutit k jejich zápisu do souboru – a to dokonce do libovolného souboru, což je vždy zásadně špatné! Okruh volajícího kódu můžeme omezit požadováním příslušných bezpečnostních oprávnění:

```
[SecurityPermissionAttribute(SecurityAction.Demand,
 SerializationFormatter=true)]
```

Role izolovaného úložiště

U některých scénářů je vhodné zvážit namísto klasických souborových I/O operací (vstupně-výstupních) raději izolované úložiště. Takovéto izolované úložiště má svoje výhody – můžeme v něm izolovat data podle uživatele a sestavení nebo podle uživatele, domény a sestavení. V prvním scénáři se do izolovaného úložiště zpravidla ukládají uživatelská data, používaná v různých aplikacích, jako je například uživatelské jméno. Uvedený postup si ukážeme v následujícím fragmentu kódu C#:

```
using System.IO.IsolatedStorage;
...
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(
    IsolatedStorageScope.User | IsolatedStorageScope.Assembly,
    null, null);
```

Druhý scénář – tedy izolace podle uživatele, domény a sestavení – zajišťuje, že se k izolovaným datům dostane jen kód z daného sestavení, a to za současného splnění následujících podmínek: jestliže dané sestavení používá stejný aplikace, jaká původně toto úložiště vytvořila, a jestliže pod ní pracuje stejný uživatel, pro kterého aplikace toto úložiště vytvořila. Vytvoření popsaného objektu znázorňuje následující ukázka kódu v jazyce Visual Basic .NET:

```
Imports System.IO.IsolatedStorage
...
Dim isoStore As IsolatedStorageFile
isoStore = IsolatedStorageFile.GetStore( _
    IsolatedStorageScope.User Or _
    IsolatedStorageScope.Assembly Or _
    IsolatedStorageScope.Domain, _
    Nothing, Nothing)
```

Poznamenejme, že izolované úložiště podporuje také „pohyblivé“ neboli cestovní profily – stačí jednoduše doplnit příznak *IsolatedStorageScope.Roaming*. Cestovní profily (roaming profiles) jsou zvláštním prvkem systému Microsoft Windows (jsou k dispozici pod Windows NT, Windows 2000 a pod některými aktualizovanými Windows 98), se kterým mohou data „následovat svého uživatele“ i po (fyzickém) přechodu k jinému počítači.

Poznámka: K izolovanému úložišti je možné přistupovat také pomocí metod `IsolatedStorageFile.GetUserStoreForAssembly` a `IsolatedStorageFile.GetUserStoreForDomain`; tyto metody neumí ale pracovat s cestovními profily.

Hlavní výhodou izolovaného úložiště například proti obyčejné třídě `FileStream` je, že příslušný kód ke své činnosti nepotřebuje oprávnění `FileIOPermission`.

Do izolovaného úložiště neukládejte ale žádná citlivá data, jako jsou šifrovací klíče a hesla, protože toto úložiště není nijak chráněno před přístupem vysoce důvěryhodného kódu, neřízeného kódu nebo také důvěryhodných uživatelů počítače.

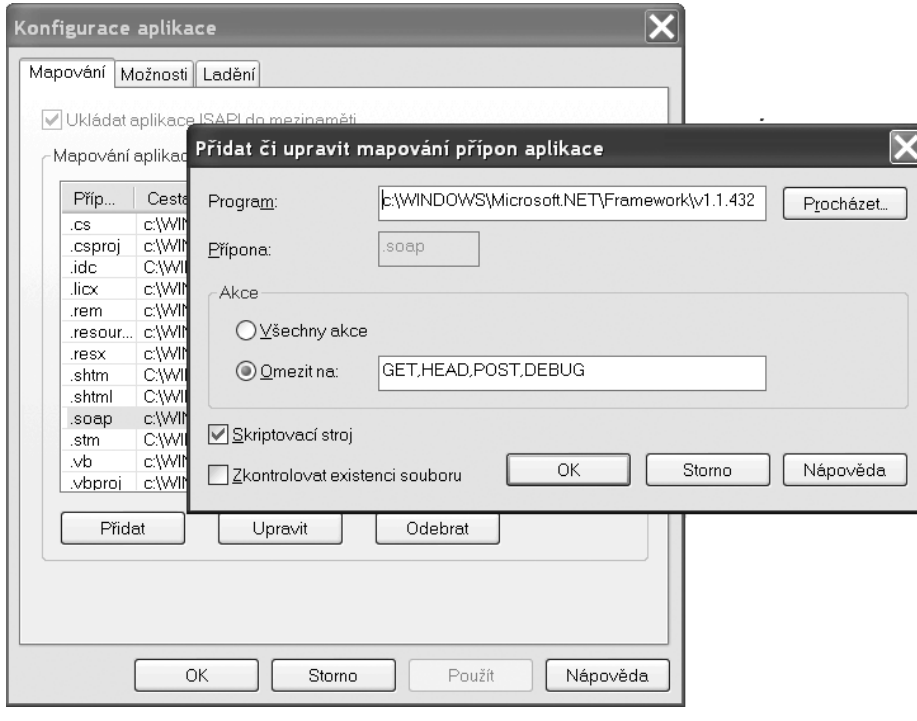
Také XSLT je kódem!

I když transformace XSLT nejsou zvláštností systému .NET Framework, hojně se zde používají a podporuje je obor názvů `System.Xml.Xsl`. Na první pohled se zdá, že transformace XSLT není ničím jiným než jazyk pro definici šablon stylů, ale ve skutečnosti se jedná o speciální programovací jazyk. Také soubory XSLT byste proto měli podrobit stejně důkladnému testování jako každý jiný skript nebo modul s kódem a měli byste také prověřit jejich chování při škodlivém vstupu, jako jsou například nepředpokládané typy dokumentů XML.

Před nasazením aplikací ASP.NET vypněte trasování a ladění

Zakázat před nasazením aplikace ASP.NET veškeré trasování a ladění je na první pohled jasné, ale byli byste překvapeni, kolik lidí to nedělá. Nevypnout tyto dvě volby je špatné hned ze dvou důvodů: jednak tím útočníkovi můžeme dát příliš mnoho informací, a jednak tyto volby mívají negativní vliv na rychlost aplikace.

Uvedené volby můžeme zablokovat třemi různými způsoby. První spočívá v odstranění klíčového slova DEBUG z parametrů IIS (Internet Information Services); v nástroji pro správu IIS najdeme tuto volbu podle obrázku 18.6.



Obrázek 18.6 Klíčové slovo DEBUG odeberte pro přípony všech typů souborů, které nebudete chtít ladit – zde jsou to soubory SOAP

Trasování a ladění je možné vypnout také ze samotné aplikace ASP.NET, kde do odpovídajících stránek doplníme zhruba takovouto direktivu `Page`:

```
<%@ Page Language="VB" Trace="False" Debug="False" %>
```

A nakonec můžeme potlačit trasování a ladění v konfiguračním souboru dané aplikace:

```
<trace enabled = 'false' />
<compilation debug = 'false' />
```

Na dálku nevypisujte podrobné chybové informace

Výchozím nastavením konfigurační volby `<customErrors>` v ASP.NET je hodnota *remoteOnly*, která znamená, že se lokálně vypisují podrobné informace a na dálku žádné. Vývojáři tento parametr na zkušebních serverech často mění – usnadní si tím přímé ladění – ale před nasazením aplikace do ostrého provozu jej zapomenou vrátit na původní hodnotu. Parametr by měl obsahovat buďto výchozí hodnotu *remoteOnly* nebo *On*; hodnota *Off* je pro provozní servery nepřípustná.

```
<configuration>
   <system.web>
      <customErrors>
         defaultRedirect="error.htm"
         mode="RemoteOnly"
         <error statusCode="404"
            redirect="404.htm"/>
      </customErrors>
   </system.web>
</configuration>
```

Deserializace dat z nedůvěryhodných zdrojů

Neprovádějte deserializaci dat z nedůvěryhodných zdrojů. To je opět naše známé rčení „veškerý vstup je zlo, dokud o něm neprokážeme opak“, jenom „přeložené“ do řeči .NET. Společná běhová knihovna .NET CLR nabízí totiž zvláštní obor názvů `System.Runtime.Serialization`, který sestavuje balíčky objektů a opět je rozbaluje, a to v procesu nazývaném *serializace*. (Někteří lidé jízlivě říkají „*sušení mrazem*“, freeze-drying.) Aplikace by ale nikdy neměla deserializovat neboli rekonstruovat data z nedůvěryhodného zdroje, protože výsledný obnovený objekt se bude na lokálním počítači provádět se stejnou úrovní důvěryhodnosti jako samotná aplikace.

Pro úspěšné vedení popsaného útoku musí mít přijímající kód také oprávněné `SerializationFormatter` – to je ovšem vysoce privilegované oprávnění, které je vhodné udělovat jen plně důvěryhodnému kódu.

Poznámka: Bezpečnostní problém, který vzniká při deserializaci dat z nedůvěryhodných zdrojů, není jen specialitou prostředí .NET – existuje i v jiných technologiích. Také knihovna MFC umožňuje například uživatelům serializaci a deserializaci objektů, a to pomocí zápisu `CArchive::Operator>>` a `CArchive::Operator<<`. Jestliže navíc uvážíme, že veškerý kód v prostředí MFC je neřízený, musí ze své podstaty pracovat jako plně důvěryhodný kód.

Při havárii neprozrazujte útočníkovi zbytečně mnoho informací

Prostředí .NET nabízí při havárii kódu a při vyvolání výjimky přímo báječné ladicí informace. Tyto údaje může ale velice snadno zneužít i útočník, který podle nich zjistí důležité informace o serverové aplikaci a o serverovém prostředí, a tím pádem může vést další útok. Příkladem může být výpis zásobníku, jenž zachytíme třeba v takovémto kódu:

```
try {
    // Tady něco uděláme
} catch (Exception e) {
    Result.WriteLine(e.ToString());
}
```

Jako výsledek se uživateli odešlou informace následujícího tvaru:

```
System.Security.SecurityException: Request for the permission of type
    System.Security.Permissions.FileIOPermission ...
    at System.Security.SecurityRuntime.FrameDescHelper(...)
    at System.Security.CodeAccessSecurityEngine.Check(...)
    at System.Security.CodeAccessSecurityEngine.Check(...)
    at System.Security.CodeAccessPermission.Demand()
    at System.IO.FileStream..ctor(...)
    at Perms.ReadConfig.ReadData() in
        c:\temp\perms\perms\class1.cs:line 18
```

Poznamenejme, že číslo řádku se odesílá pouze v ladicí verzi celé aplikace. Uvedené informace jsou ale natolik rozsáhlé a zároveň natolik „intimní“, že by je neměl dostat do rukou nikdo než vývojáři a testeři, kteří pracují na tomto kódu. Při vyvolání výjimky proto jednoduše zapíšeme odpovídající záznam do protokolu událostí Windows a uživateli odešleme zprávu, že požadavek selhal.

```
try {
    // Tady něco uděláme.
} catch (Exception e) {
    #if(DEBUG)
        Result.WriteLine(e.ToString());
    #else
        Result.WriteLine("Došlo k chybě.");
        new LogException().Write(e.ToString());
    #endif
}

public class LogException {
    public void Write(string e) {
        try {
            new EventLogPermission(
                EventLogPermissionAccess.Instrument,
                "machinename").Assert();
            EventLog log = new EventLog("Application");
            log.Source="MyApp";
            log.WriteEntry(e, EventLogEntryType.Warning);
        } catch(Exception e2) {
        // Auu! Nemůžu zapisovat do protokolu událostí.
        }
    }
}
```

Podle povahy konkrétní aplikace je někdy nutné provést volání `EventLogPermission(...).Assert()`, jako je tomu v předcházející ukázce kódu. Samozřejmě že pokud aplikace nemá oprávnění k zápisu do protokolu událostí, dojde k vyvolání další výjimky.

Shrnutí

Systém .NET Framework a běhová knihovna CLR nabízí řešení celé řady bezpečnostních problémů. Z nich nejdůležitější asi je, že řízené prostředí pomáhá při potlačení problémů s přetečením bufferu v uživatelských aplikacích a že zajišťuje bezpečnost přístupového kódu a pomáhá tím řešit rozhodování mezi důvěryhodným, částečně důvěryhodným a nedůvěryhodným kódem. To ale neznamená, že bychom mohli usnout na vavřínech: nezapomeňte, že každý kód se někdy stane cílem útoku, a že se v něm proto musíte bránit.

Velká část rad uvedených v této knize platí i pro řízené aplikace: neukládejte tajné informace do webových stránek a do kódu, spouštějte aplikace s nejmenšími možnými oprávněními a žádejte vždy jen o omezenou množinu oprávnění a dále buďte velmi opatrní, pokud provádíte nějaké rozhodnutí jen na základě názvu něčeho. Dále je vhodné převést všechny ovládací prvky ActiveX do řízeného kódu a nové ovládací prvky již vytvářet jen v řízeném kódu; zkrátka a dobře, řízený kód je bezpečnější.

A nakonec, Microsoft sám nabízí na stránkách *http://msdn.microsoft.com/* celou řadu dokumentů souvisejících s bezpečností .NET. Podívejte se zejména na dokument „Security Concerns for Visual Basic .NET and Visual C# .NET Programmers“, který se nachází na adrese *http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vbtch-SecurityConcernsForVisualBasicNETProgrammers.asp* a ze kterého se dostanete k těm nejdůležitějším informacím.