APP LOCKDOWN

Ochrona aplikacji i ważnych danych użytkownika poprzez stosowanie technik kodowania defensywnego

Kenny Kerr

W artykule tym omówiono następujące tematy:

- Zabezpieczanie poświadczeń tożsamości użytkownika oraz informacji logowania.
- Ochrona danych klienta i serwera poprzez szyfrowanie.
- Ochrona klientów COM i serwerów COM.

Artykuł dotyczy następujących technologii:
C++, COM, bezpieczeństwo, C#, .NET

Przykłady kodu do pobrania:
AppLockdown.exe (165KB)

W dzisiejszym świecie, w którym wszystko połączone jest ze wszystkim, a każda aplikacja jest potencjalnym celem ataku, technikami programowania defensywnego należy objąć także zabezpieczenia. Umiejętność programowania defensywnego pomaga nam w pisaniu bardziej bezpiecznego kodu, ale to ciągle nie wystarczy. Trzeba pójść o wiele dalej i do tworzonego oprogramowania wprowadzić jawną ochronę.

W tym artykule skupimy uwagę na ochronie użytkowników, zabezpieczaniu ich poświadczeń tożsamości oraz danych prywatnych, a także na ochronie serwerów. Omówiony zostanie również szeroki zakres często stosowanych scenariuszy programistycznych oraz praktycznych sposobów pisania kodu bardziej odpornego na ataki.

Ochrona poświadczeń tożsamości

Zacznijmy od tego, w jaki sposób można chronić poświadczenia tożsamości użytkowników. Aby poświadczenia tożsamości były dobrze chronione, należy unikać jawnego zarządzania poświadczeniami tożsamości użytkowników, ponieważ hasła powinny być trzymane w tajemnicy, a tajemnice trudno zabezpieczyć. Dlatego należy wykorzystywać sesje logowania i pojedyncze logowanie. Jeśli istnieje potrzeba uruchomienia aplikacji z uprawnieniami innego użytkownika, należy skorzystać z opcji *Uruchom jako* lub wpisać polecenie *runas* w wierszu polecenia. Jeśli jednak musimy sami zarządzać poświadczeniami, to powinniśmy zrobić to w sposób opisany poniżej.

Aby móc używać funkcji opisanych w tym dokumencie, należy zainstalować Platform SDK w wersji z lutego 2003 roku lub nowszej. Po zainstalowaniu pakietu należy dodać katalogi Platform SDK o nazwie *Include* i *Lib* do ścieżek wyszukiwania Visual C++®. Jeśli wolimy stosować wywołania P/Invoke z C# lub z Visual Basic® .NET, to nie musimy instalować pakietu Platform SDK.

Następnym krokiem jest poproszenie użytkownika o podanie poświadczeń. Aby zapewnić spójny interfejs użytkownika i nie musieć pisać dużych ilości kodu związanego z bezpieczeństwem, w przypadku aplikacji

z graficznym interfejsem użytkownika (GUI) należy stosować funkcję CredUIPromptForCredentials, a w przypadku aplikacji konsolowych — funkcję CredUICmdLinePromptForCredentials. Obydwie funkcje mają praktycznie tę samą funkcjonalność, ale wersja z graficznym interfejsem użytkownika ma więcej opcji — umożliwia na przykład zmianę domyślnej bitmapy banera, etykiet i komunikatów w oknie dialogowym. Funkcje te można wykorzystać do uzyskania od użytkownika poświadczeń tożsamości w domenie Windows lub poświadczeń ogólnych, co pokazano na **ilustracji 1**.

Przy zapisywaniu poświadczeń tożsamości w profilu użytkownika jako klucz stosowana jest nazwa docelowa (target name), dlatego warto stosować znaczące i unikalne nazwy docelowe. Struktura CREDUI_INFO umożliwia podanie uchwytu do okna rodzica dla modalnego okna dialogowego. Pozwala także na zastąpienie domyślnego tekstu etykiet i komunikatu oraz bitmapy banera.

Jest jeszcze problem przepełnienia całkowitoliczbowego. Na ilustracji można zauważyć dwie operacje rzutowania static_cast. Zostały zastosowane po to, by przekonwertować wynik wykonania metody std::vector<wchar_t>::size na wartość typu ULONG, ponieważ takiego typu parametru oczekuje funkcja CredUIPromptForCredentials. Chociaż wartość ULONG zawsze będzie 32-bitową liczbą całkowitą bez znaku, to size_t jest na tyle duże, by objąć cały zakres wskaźnika. Innymi słowy, size_t jest 32-bitową liczbą całkowitą bez znaku na 32-bitowej wersji systemu Windows, a na 64-bitowej wersji systemu Windows jest 64-bitową liczbą bez znaku. Rzutowanie static_cast stosuje się po to, by powstrzymać ostrzeżenia kompilatora sygnalizujące, że rzutowanie to może nie być bezpieczne w 64-bitowej wersji Windows. Powstrzymując ostrzeżenia kompilatora programista bierze na siebie pełną odpowiedzialność za bezpieczeństwo rzutowania — nawet na platformach 64-bitowych. W poprzednim przykładzie widać, że wielkość wektora jest mniejsza niż std::numeric_limits<ULONG>::max(), więc w tym przypadku rzutowanie jest bezpieczne. Jak widać, błędy można popełnić na każdym kroku, dlatego napisałem klasę PromptForCredentials, aby uprościć i zabezpieczyć zarządzanie poświadczeniami tożsamości. Klasa ta jest dostępna wraz z przykładowym kodem dołączonym do tego artykułu. W poniższym przykładzie zaprezentowano sposób wykorzystania klasy PromptForCredentials.

```
PromptForCredentials prompt;
prompt.Target(L"server");
prompt.ParentWindow(0); // przypisanie do okna-rodzica
prompt.Flags(CREDUI_FLAGS_GENERIC_CREDENTIALS);
if (prompt.ShowDialog()) {
    // TODO: wykorzystać prompt.UserName() oraz prompt.Password()
    prompt.ScrubPassword();
}
```

Przykładowy kod jest prosty i zajmuje się także obsługą błędów. Destruktor klasy PromptForCredentials automatycznie wywoła metodę ScrubPassword, ale dobrym rozwiązaniem jest wyzerowanie wszystkich tajnych danych w pamięci tak szybko, jak to tylko możliwe. Wewnętrznie metoda ScrubPassword wywołuje SecureZeroMemory (będącą substytutem funkcji RtlSecureZeroMemory) w celu wyzerowania hasła znajdującego się w buforze.

Ilustracja 2. Prośba o podanie poświadczeń tożsamości

Aby ułatwić eksperymenty z wieloma opcjami i flagami dostępnymi w funkcji CredUIPromptForCredentials, napisałem przykładową aplikację o nazwie „Prompt for Credentials", widoczną na **ilustracji 2**. Aplikacja ta pozwala na zastosowanie różnych opcji i flag, co umożliwia ustalenie opcji wymaganych do uzyskania poszukiwanej funkcjonalności.

Zabezpieczanie danych po stronie klienta

Ochrona ważnych danych związana jest z szyfrowaniem danych znajdujących się w pamięci oraz podczas zapisywania ich na dysku. Jeśli dane są trwale przechowywane, to muszą być chronione silną listą kontroli dostępu (ACL). Z tej sekcji dowiemy się, w jaki sposób poprawnie wykonywać te zadania.
Tworzenie kodu szyfrującego za pomocą bezpłatnych i dobrych bibliotek kryptograficznych, takich jak CryptoAPI, CAPICOM i System.Security.Cryptography, jest dosyć łatwym zadaniem, ale zarządzanie kluczami szyfrowania i hasłami nadal pozostaje wyzwaniem. Windows XP oraz Windows Server 2003 zawierają funkcje ochrony danych, umożliwiające szyfrowanie i deszyfrowanie danych bez konieczności zarządzania kluczami szyfrowania. Do szyfrowania danych można użyć funkcji CryptProtectData. Dane można później odszyfrować wyłącznie za pomocą funkcji CryptUnprotectData wywołanej w sesji logowania tego samego użytkownika. Odpowiedni przykład przedstawiono na **ilustracji 3**.

CheckError to funkcja pomocnicza, sprawdzająca wartość zwracaną przez funkcję. Jeśli zwracana wartość oznacza porażkę, zgłaszany jest odpowiedni wyjątek. Funkcję ChceckError oraz wiele innych przydatnych funkcji i klas można znaleźć w pliku udostępnionym wraz z tym artykułem. Jak widać, nie ma potrzeby zarządzania kluczem szyfrującym. Po pierwsze, od użytkownika należy pobrać dane do zaszyfrowania — na przykład wczytać tekst z kontrolki do bufora za pomocą funkcji GetWindowTextLength oraz GetWindowText. Następie wypełniana jest struktura DATA_BLOB, wskazująca funkcji CryptProtectData położenie i rozmiar danych zapisanych w postaci jawnej. Wewnętrznie funkcja CryptProtectData za pomocą funkcji LocalAlloc alokuje ilość pamięci wystarczającą do zapisania w niej zaszyfrowanych danych i w innej strukturze DATA_BLOB zwraca wskaźnik i rozmiar tego obszaru pamięci. Teraz można przystąpić do zapisania tajnych danych na dysku (jeśli jest to konieczne) i zwolnić pamięć przy użyciu funkcji LocalFree (należy także pamiętać o jak najszybszym wymazaniu danych jawnych za pomocą funkcji SecureZeroMemory). Odszyfrowywanie dokonywane jest za pomocą funkcji CryptUnprotectData, która jest bardzo podobna do funkcji CryptProtectData. Pierwszy parametr wskazuje strukturę DATA_BLOB odwołującą się do tekstu zaszyfrowanego,

a ostatni parametr wskazuje strukturę DATA_BLOB, dla której funkcja zaalokuje pamięć przy użyciu funkcji LocalAlloc i zapisze w niej dane w postaci zdeszyfrowanej.
Zarządzanie strukturami DATA_BLOB jest zadaniem uciążliwym i mogącym doprowadzić do błędów, jeśli zapomnimy zwolnić i wyczyścić w odpowiednim momencie pamięć. Trudne jest również stosowanie tej struktury z kodem obsługi wyjątków. Aby uprościć te zagadnienia, można wykorzystać klasę DataBlob widoczną na **ilustracji 4**.

Destruktor klasy DataBlob automatycznie wyczyści i zwolni pamięć, gdy będzie to potrzebne. Jest to doskonałe rozwiązanie na bezpieczne przechowywanie danych w systemie plików, jeśli jednak konieczne jest tylko zabezpieczenie danych znajdujących się w pamięci podczas działania aplikacji, to rozwiązanie to zużywa zbyt wiele pamięci. Dane znajdujące się w przestrzeni adresowej procesu należy szyfrować, ponieważ możliwe jest zapisanie stron z tej przestrzeni do pliku wymiany lub do pliku hibernacyjnego. Jeśli atakujący może naruszyć inne aspekty zabezpieczeń komputera — może na przykład uzyskać do niego fizyczny dostęp — to może odczytać tajne dane.

Właśnie z tego powodu w Windows Server 2003 wprowadzono funkcje CryptProtectMemory i CryptUnprotectMemory. W Windows 2000 i Windows XP w tym samym celu można stosować funkcje RtlEncryptMemory oraz RtlDecryptMemory, ale należy zachować ostrożność, ponieważ funkcje te mogą nie być dostępne na nowszych platformach. Funkcje CryptProtectMemory i CryptUnprotectMemory zaprojektowano tak, by wydajnie szyfrowały blok pamięci w miejscu (in-place). Jest to przydatne rozwiązanie, jeśli jakieś ważne dane (jak na przykład poświadczenia tożsamości) mają być przechowywane w pamięci przez dłuższy czas lub mają być stosowane wielokrotnie. Funkcje te mają ciekawą cechę — dane do zaszyfrowania muszą zajmować wielokrotność określonego bloku. Algorytmy szyfrujące zazwyczaj szyfrują strumień danych blok po bloku. Dane krótsze niż blok muszą zostać dopełnione tak, by zajmowały cały blok. Dlatego jeśli chcemy szyfrować jakiekolwiek dane, powinniśmy upewnić się, że rozmiar bufora, w którym dane te są przechowywane, jest wielokrotnością rozmiaru bloku. W poniższym przykładzie przedstawiono prostą funkcję wyznaczającą dla bufora rozmiar będący wielokrotnością rozmiaru bloku:

```
DWORD ToBlockSize(DWORD original) {
    DWORD result = CRYPTPROTECTMEMORY_BLOCK_SIZE;
    if (0 != original) {
        DWORD remainder = original % CRYPTPROTECTMEMORY_BLOCK_SIZE;
        result = original + (0 != remainder ?
            CRYPTPROTECTMEMORY_BLOCK_SIZE - remainder : 0);
    }
    return result;
}
```

Dzięki powyższej funkcji stosowanie funkcji CryptProtectMemory staje się łatwe. W poniższym przykładzie pokazano, jak odczytać tekst z kontrolki w aplikacji MFC i zaszyfrować go:

```
CWnd* pControl = GetDlgItem(IDC_SECRET);
DWORD charCount = pControl->GetWindowTextLength() + 1;

std::vector<BYTE> buffer(ToBlockSize(charCount * sizeof (WCHAR)));
pControl->GetWindowText(reinterpret_cast<PWSTR>(&buffer[0]), charCount);

if (!::CryptProtectMemory(&buffer[0], static_cast<DWORD>(buffer.size()),
        CRYPTPROTECTMEMORY_SAME_PROCESS)) {
    AtlThrowLastWin32();
}
```

Odszyfrowanie danych przeprowadza się poprzez wywołanie funkcji CryptUnprotectMemory z dokładnie takim samym zestawem parametrów. W buforze znowu znajdą się dane jawne.

Jak na razie, mówiliśmy o szyfrowaniu danych przy długo- i krótkoterminowym ich przechowywaniu. Pozostała jeszcze kwestia zabezpieczenia danych podczas przechowywania ich w systemie plików lub w rejestrze. Profil użytkownika, zawierający folder *Moje dokumenty*, zapewnia bezpieczne miejsce, w którym aplikacje mogą zapisywać związane z użytkownikiem dane oraz jego dokumenty. W profilu użytkownika znajduje się także część rejestru bieżącego użytkownika oraz gałąź systemu plików. Dane te zabezpieczone są listami ACL, które skutecznie ograniczają dostęp do danych użytkownika (prawo dostępu ma tylko dany użytkownik i grupa *Administratorzy*). Dwa najbardziej użytecznie foldery w profilu użytkownika to folder *Dane aplikacji* oraz folder osobisty o nazwie *Moje dokumenty*. Folder *Dane aplikacji* służy do zapisywania plików zawierających dane aplikacji związane z konkretnym użytkownikiem, ale których użytkownik nie używa bezpośrednio. W folderze tym zapisywane są na przykład pliki konfiguracyjne, zawierające preferowane ustawienia danego użytkownika. Natomiast folder *Moje dokumenty* powinien być lokalizacją domyślną, podpowiadaną przez aplikacje przy zapisywaniu utworzonych przez użytkownika plików.

Aby użyć tych folderów w kodzie programu pisanego w języku C++, należy wykorzystać funkcję SHGetFolderPathAndSubDir. Funkcjonalność tej wprowadzonej w Windows XP funkcji odpowiada połączonej funkcjonalności wywoływanych kolejno funkcji SHGetFolderPath i PathAppend. Jeśli folder nie istnieje, to funkcja może także ten folder utworzyć. Oto przykład jej działania:

```
std::vector<wchar_t> path(MAX_PATH);
CheckError(::SHGetFolderPathAndSubDir(0 /*brak okna-rodzica */,
    CSIDL_FLAG_CREATE | CSIDL_APPDATA, 0 /*brak tokenu */,
    SHGFP_TYPE_CURRENT, L"Kerr\\SecuritySample", &path[0]));
::PathAppend(&path[0], L"settings.xml");
CHandle file(::CreateFile(&path[0], ...));
```

Ponieważ funkcje powłoki Windows nie przyjmują argumentu z rozmiarem bufora, trzeba zwrócić szczególną uwagę na zabezpieczenie aplikacji przed błędem przepełnienia bufora. Aby utworzyć plik w folderze *Moje dokumenty*, należy zamienić flagę CSIDL_APPDATA na CSIDL_PERSONAL. Równoważny kod w języku C# jest trochę mniej spójny, ale prostszy i mniej podatny na błędy:

```
string path =
    Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
path = Path.Combine(path, @"Kerr\SecuritySample");
Directory.CreateDirectory(path);
path = Path.Combine(path, "settings.xml");
FileStream file = new FileStream(path, ...);
```

Ochrona klientów strumieni nazwanych

Strumienie nazwane to jedna z najprostszych i najbardziej wydajnych form komunikacji międzyprocesowej (IPC) na platformie Windows. Strumień nazwany to proste jednokierunkowe lub dwukierunkowe połączenie, za pośrednictwem którego klient i serwer mogą przesyłać komunikaty lub strumienie bajtów. Strumienie nazwane zapewniają wyższy poziom abstrakcji niż gniazda i interfejs SSPI (Security Service Provider Interface), co ułatwia programowanie komunikacji sieciowej. Istnieją jednak pewne drobne problemy związane ze sposobem uwierzytelniania stosowanym przez strumienie.

W tej sekcji zajmiemy się nazwanymi strumieniami z perspektywy klienta strumienia nazwanego. Nawet jeśli aplikacja nie wykorzystuje bezpośrednio strumieni nazwanych, to może być od nich uzależniona poprzez jakiś interfejs API. Strumienie nazwane są na przykład wykorzystywane przez funkcje menedżera sterowania usługami do zarządzania lokalnymi i zdalnymi usługami systemu Windows. Zdalny dostęp do rejestru z zastosowaniem funkcji RegConnectRegistry także oparty jest na strumieniach nazwanych. Dobre zrozumienie zabezpieczenia strumieni nazwanych ułatwia zrozumienie sposobu działania wielu popularnych usług Windows, a także własnych aplikacji.
Klient podłącza się do strumienia nazwanego przy użyciu funkcji CreateFile. Powoduje to wewnętrzną delegację zarządzania komunikacją z serwerem strumienia nazwanego do usługi *Stacja robocza* (Workstation). Rozważmy następujący przykład:

```
CHandle handle(::CreateFile(L"\\\\server\\pipe\\Kerr.SecuritySample",
    FILE_READ_DATA | FILE_WRITE_DATA, 0 /*brak współdzielenia */,
    0 /*domyślne atrybuty bezpieczeństwa */, OPEN_EXISTING,
    SECURITY_SQOS_PRESENT | SECURITY_IDENTIFICATION, 0 /*bez szablonu */));
```

Żądając prawa zapisu i odczytu ze strumienia, klient przyjmuje, że serwer utworzył strumień w trybie dupleks lub w trybie dwukierunkowym. Jeśli jest inaczej, to funkcja CreateFile nie zakończy się pomyślnie i funkcja GetLastError zwróci błąd o kodzie ERROR_ACCESS_DENIED.

Kolejną interesującą sprawą są informacje jakości usług zabezpieczeń (Security Quality of Service — SQOS). W komunikacji za pośrednictwem strumieni nazwanych zazwyczaj stosuje się flagę SECURITY_SQOS_PRESENT w połączeniu z SECURITY_IDENTIFICATION lub SECURITY_IMPERSONATION. SECURITY_IMPERSONATION jest ustawieniem domyślnym, jeśli nie podaje się żadnych informacji o SQOS. Jednym z głównych celów programisty tworzącego aplikację kliencką powinna być ochrona tożsamości klienta. Oznacza to ograniczenie zaufania do serwerów, z którymi aplikacja się komunikuje.

Flaga SECURITY_IMPERSONATION oznacza, że serwer będzie mógł uzyskać dostęp do lokalnych zasobów w imieniu klienta. Na przykład serwer może wcielić się w klienta po to, by przy wywołaniu funkcji CreateFile

sprawdzenie praw dostępu do systemu plików przeprowadzone zostało na podstawie tożsamości klienta. Jeśli nie chcemy, by serwer wcielał się w klienta, możemy podać flagę SECURITY_IDENTIFICATION. Flaga ta oznacza, że serwer będzie mógł podać się za klienta w celu uzyskania informacji uwierzytelniających klienta, ale nie będzie mógł podać się za klienta przy uzyskiwaniu dostępu do innych zasobów.
Ważny jest także sposób uwierzytelniania klienta. Funkcja CreateFile nie pozwala na podanie poświadczeń tożsamości, dlatego musi istnieć inny sposób uwierzytelniania komunikacji z serwerem strumienia nazwanego — tym bardziej, że dość często potrzebne jest podanie poświadczeń alternatywnych. Jest to łatwiej zrozumieć jeśli zauważymy, że strumienie nazwane są naprawdę obsługiwane przez usługę *Serwer*, która zarządza także współdzieleniem plików i drukarek. Wspomniana wcześniej usługa *Stacja robocza* (Workstation) to po prostu usługa zarządzająca sesjami połączeń do usług *Serwer* znajdujących się na innych komputerach. Infrastrukturę tę zwyczajowo nazywa się serwerem plików Windows.

Stosowanie serwera plików może być dość trudne, ponieważ w danej sesji logowania do danego serwera może być podłączona tylko jedna sesja klienta. Próba nawiązania drugiego połączenia z innym zestawem uwierzytelnień nie powiedzie się. Dlatego zarządzanie sesjami jest ważną częścią tworzenia aplikacji klienta strumienia nazwanego.
Sesje są zazwyczaj tworzone automatycznie w imieniu użytkownika, gdy ten za pomocą programu *Eksplorator Windows* łączy się z udziałem na komputerze zdalnym. Sesje te zazwyczaj nie mają długiego okresu życia i są automatycznie zamykane przez usługę *Stacja robocza* w jakiś czas po zamknięciu uchwytu pliku przez klienta. Do nawiązania połączenia stosowana jest tożsamość z sesji logowania, w której zainicjowano połączenie. Jednak czasami musi zostać użyty inny zestaw poświadczeń.

Podczas nawiązywania połączenia ze zdalnym komputerem poświadczenia bieżącej sesji logowania mogą nie być wystarczające do nawiązania tego połączenia. Można wtedy zastąpić je alternatywnym zestawem poświadczeń. Zestaw ten nazywany jest rekordem użycia (use record). Przydatny jest przy tworzeniu sesji połączenia do serwera plików z jawnym podaniem zestawu poświadczeń, kiedy sesja logowania użytkownika albo nie posiada poświadczeń sieciowych, albo nie posiada praw dostępu do serwera zdalnego.
Aby utworzyć rekord użycia, należy skorzystać z funkcji NetUseAdd. Na **ilustracji 5** znajduje się przykład użycia zaprezentowanej wcześniej klasy PromptForCredentials. Gdy rekord użycia nie jest już dłużej potrzebny, należy usunąć go, wywołując funkcję NetUseDel:

```
CheckError(::NetUseDel(0 /*zarezerwowane*/, L"\\\\server\\IPC$", USE_FORCE));
```

Ochrona klientów COM

Szczegółowy opis architektury i zabezpieczeń COM zająłby co najmniej jedną książkę (jak nie dwie). Dlatego w tej sekcji skoncentrujemy się wyłącznie na najważniejszych aspektach zabezpieczeń COM i ochronie aplikacji klienckich COM. Architektura COM nadal jest ważna dla programistów, ponieważ jest stosowana w wielu usługach systemu Windows. Niektóre obiekty COM zostały opakowane kodem zarządzanym i są dostępne poprzez interfejsy w środowisku .NET Framework, ale nadal są komponentami COM. Nie zanosi się, aby miało to ulec zmianie w najbliższym czasie.

Aby uprościć terminologię, kod wykorzystujący obiekty COM będziemy określać jako klienty COM, a obiekty COM — jako serwery COM, bez względu na sposób ich aktywacji. Takie nazewnictwo jest zgodne ze specyfikacją COM.

Pierwsza interesująca funkcja to funkcja CoCreateInstance, wywoływana przez typowego klienta COM po wybraniu modelu wątków. Przedstawiono ją w poniższym przykładzie:

```
CheckError(::CoCreateInstance(__uuidof(CoLibraryObject),
    0 /*brak wskazania zewnetrznego interfejsu IUnknown*/, CLSCTX_INPROC_SERVER,
    __uuidof(ILibraryObject),
    reinterpret_cast<PVOID*>(&spLibraryObject)));
```

Serwer COM zostanie utworzony w procesie klienta, wskazanym przez flagę CLSCTX_INPROC_SERVER. CLCTX — czyli wartość kontekstu klasy — nie określa, gdzie obiekt zostanie aktywowany (jest to kontrolowane przez serwer). Wartość ta jest raczej stosowana w celu ograniczenia możliwych miejsc utworzenia instancji obiektu. Jeśli kontekst klasy nie pasuje do zdefiniowanego przez serwer sposobu tworzenia obiektów, wywołanie funkcji CoCreateInstance nie powiedzie się. Ponieważ serwer w tym przykładzie działa wewnątrz sesji logowania klienta, może w imieniu klienta zrobić w zasadzie wszystko. Dlatego skoncentrujemy się na aktywacji poza procesem, przedstawionej w poniższym przykładzie:

```
CheckError(::CoCreateInstance(...,
    CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER, ...));
```

Flaga CLSCTX_LOCAL_SERVER wskazuje, że klient COM aktywuje serwer COM poza procesem. Można także użyć flagi CLSCTX_ALL która wskazuje, że dla klienta COM nie ma znaczenia, gdzie będzie uruchomiony serwer COM. Jest to domyślne zachowanie metody CComPtr<T>::CoCreateInstance. Aby uniknąć wszelkich niespodzianek, należy zawsze jawnie definiować, w jaki sposób serwer COM ma być uruchamiany.

W poprzednim przykładzie klient COM przekazywał serwerowi COM tożsamość procesu klienta w celu uwierzytelnienia i autoryzacji. W takim przypadku serwer COM uzyska token dla sesji logowania klienta. Jeśli serwer COM zostałby aktywowany na zdalnym komputerze, to w celu reprezentowania klienta zostałaby utworzona sesja logowania do sieci.

Jeśli klient stosuje personifikację — co oznacza, że istnieje token wątku — i chcemy podać serwerowi COM tożsamość wątku klienta, musimy wcześniej włączyć dynamiczne zasłanianie. Można je włączyć dla całego procesu przy użyciu funkcji CoInitializeSecurity, ale często nie jest to dobre rozwiązanie, ponieważ ma wpływ na

wszystkich klientów COM w danym procesie. Lepszym podejściem jest włączenie dynamicznego zasłaniania dla konkretnego interfejsu pośredniczącego COM. Można to zrobić za pomocą funkcji CoSetProxyBlanket:

```
CheckError(::CoSetProxyBlanket(spServerObject, RPC_C_AUTHN_DEFAULT,
    RPC_C_AUTHZ_NONE, COLE_DEFAULT_PRINCIPAL,
    RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_DEFAULT,
    COLE_DEFAULT_AUTHINFO, EOAC_DYNAMIC_CLOAKING));
```

W większości parametrów podano wartości domyślne. Interesujące nas parametry to parametr pierwszy — określający klasę pośredniczącą, którą należy skonfigurować, oraz parametr ostatni — włączający funkcjonalność EOAC_DYNAMIC_CLOAKING. Przez cały czas życia klasa pośrednicząca przy każdym wywołaniu metody ustali odpowiedni token i zastosuje go przy podłączaniu się do serwera COM. Może to wydawać się rozwiązaniem nieefektywnym, ale tak nie jest. Ponieważ rozproszony COM (DCOM) używa do komunikacji protokołu RPC (remote procedure call), wewnętrznie obiekt kanału klasy pośredniczącej jest skojarzony z uchwytem powiązania RPC. Uchwyt ten zarządza informacjami uwierzytelniającymi, stosowanymi do łączenia się z określonym serwerem. W kolejnych wywołaniach — o ile token nie ulegnie zmianie — uchwyt ten wykorzystywany jest ponownie.

Innym rozwiązaniem może być zastosowanie do połączenia z komputerem zdalnym innego zestawu poświadczeń. Może to być przydatne rozwiązanie, jeśli klient nie potrafi ustanowić sesji logowania do sieci na tym komputerze, na przykład gdy komputer nie jest częścią zaufanej domeny. Jawne poświadczenia podawane są za pośrednictwem struktury COAUTHIDENTITY. Struktura ta może być użyta podczas pierwszego żądania aktywacji w celu połączenia ze zdalnym komputerem przy użyciu funkcji CoCreateInstanceEx. Struktura może być również wykorzystana wtedy, gdy przy użyciu funkcji CoSetProxyBlanket ustawiane są informacje uwierzytelniające dla klasy pośredniczącej. Dla obydwu funkcji można podać różne lub takie same uwierzytelnienia. Jeśli poświadczenia zostaną użyte podczas aktywacji, nie będą miały żadnego wpływu na klasy pośredniczące. Klasy pośredniczące będą nadal używać tokenu procesu w normalnych warunkach lub tokenu efektywnego gdy włączone jest dynamiczne zasłanianie. Jednak bez względu na to, czy stosowane są różne, czy takie same poświadczenia dla aktywacji i dla klasy pośredniczącej, zawsze tworzone są dwie niezależne sesje logowania.
Aby uchronić się od częstych błędów związanych z przepełnieniem bufora, napisałem widoczną na **ilustracji 6** klasę CoAuthIdentity do wypełniania struktur COAUTHIDENTITY. Klasa ta zajmuje się obsługą błędów, które mogą wystąpić przy korzystaniu z buforów łańcuchów znaków — zwłaszcza wtedy, gdy stosowane są dla haseł. Wywołanie funkcji CoCreateInstanceEx jest teraz dosyć proste, co widać na **ilustracji 7**.

Uwierzytelnień (a w szczególności haseł) nie należy zapisywać bezpośrednio w kodzie, jak zrobiłem to w przykładach. W przykładach dla tego artykułu podałem hasła w kodzie programu, ponieważ upraszcza to przykłady i sprawia, że są bardziej związane z tematem.

Do funkcji CoCreateInstanceEx przekazywana jest struktura COSERVERINFO, która w strukturze COAUTHINFO zawiera nazwę komputera zdalnego oraz informacje uwierzytelniające. W podanym przykładzie struktura COAUTHINFO została ustawiona tak, by uwierzytelnianie odbywało się za pośrednictwem Windows NT LAN

Manager (NTLM). Jest to bezpieczne rozwiązanie, ponieważ NTLM jest zawsze dostępne. Jedyny powód, dla którego można nie chcieć stosować NTLM, to niemożliwość weryfikacji tożsamości serwera, co zostanie omówione w dalszej części tego artykułu.

Ponieważ do zdalnej aktywacji funkcja CoCreateInstanceEx w zakresie komunikacji ze zdalnym SCM zdaje się na lokalne COM SCM, konieczne jest zastosowanie RPC_C_IMP_LEVEL_IMPERSONATE jako poziomu personifikacji. SCM musi posiadać poziom zaufania umożliwiający zainicjowanie w naszym imieniu zdalnej sesji logowania.

Jak już wspomniałem, uwierzytelnienia zastosowane przy żądaniu aktywacji nie mają wpływu na klasę pośredniczącą, która zostanie zwrócona przez funkcję CoCreateInstanceEx. Aby uchwyt wiązania klasy pośredniczącej uwierzytelniał się za pomocą tych samych poświadczeń, do metody CoSetProxyBlanket należy przekazać strukturę COAUTHIDENTITY lub obiekt CoAuthIdentity, tak jak zrobiono to w poniższym przykładzie:

```
CheckError(::CoSetProxyBlanket(..., &authIdentity, EOAC_DEFAULT));
```

Należy pamiętać, że klasa pośrednicząca — dopóki nie zostanie zwolniona — może odwoływać się do struktury COAUTHIDENTITY w dowolnym momencie, dlatego ważne jest, by pamięć nie została zmieniona lub zwolniona przed zwolnieniem wszystkich klas pośredniczących.

Dotąd zajmowaliśmy się sposobami zarządzania uwierzytelnianiem klientów COM przez serwery COM. Kolejnym ważnym aspektem bezpieczeństwa COM jest poziom zaufania zastosowany na serwerze. Poprzez połączenie z serwerem COM, klient COM umożliwia serwerowi personifikację klienta. Serwer może zrobić z otrzymanym tokenem wątku tylko to, na co zezwala klient. Chociaż określenie praw możliwe jest na poziomie komputera lub procesu, zalecane jest, by zawsze określać prawa dla klas pośredniczących. Wróćmy do funkcji CoSetProxyBlanket, pokazanej w poniższym przykładzie:

```
CheckError(::CoSetProxyBlanket(..., RPC_C_IMP_LEVEL_IMPERSONATE, ...));
```

Odpowiednimi wartościami dla parametru dwImpLevel są: RPC_C_IMP_LEVEL_IDENTIFY, RPC_C_IMP_LEVEL_IMPERSONATE oraz RPC_C_IMP_LEVEL_DELEGATE.

Najpierw zajmijmy się opcją RPC_C_IMP_LEVEL_DELEGATE. Delegacja to możliwość przyjęcia przez serwer poświadczeń tożsamości klienta i użycia ich do ustanowienia kolejnej sesji logowania na innym komputerze. Oczywiście rozwiązanie to będzie działać wyłącznie wtedy, gdy uwierzytelnienia klienta są buforowane przez sesję logowania. NTLM nie obsługuje delegacji, ale obsługuje ją Kerberos. Jeśli klient stosuje flagę RPC_C_IMP_LEVEL_DELEGATE w celu uzyskania dostępu do lokalnego serwera COM, delegacja będzie obsługiwana nawet wtedy, gdy do lokalnego uwierzytelniania wykorzystywany jest NTLM. Jest to możliwe, ponieważ przy połączeniu lokalnego klienta COM z lokalnym serwerem COM wykorzystywana jest sesja logowania klienta i nie jest tworzona sesja logowania sieciowego (tak jak w przypadku podłączania się do zdalnych serwerów COM). Delegacja działa skutecznie, ponieważ sesja logowania klienta zazwyczaj zawiera uwierzytelnienia konieczne do komunikacji sieciowej. Mając na uwadze możliwości, jakimi dysponuje serwer posiadający delegowane uwierzytelnienia klienta, należy starannie rozważyć zastosowanie takiego rozwiązania.

Flaga RPC_C_IMP_LEVEL_IDENTIFY działa tak samo jak flaga SECURITY_IDENTIFICATION stosowana przez klientów strumieni nazwanych. Serwer może wykorzystać token klienta w celu zidentyfikowania klienta i dokonania uwierzytelnienia. Serwer nie będzie jednak mógł personifikować klienta w celu uzyskania dostępu do zabezpieczonych zasobów, takich jak system plików.

Flaga RPC_C_IMP_LEVEL_IMPERSONATE ma takie samo działanie jak flaga SECURITY_IMPERSONATION, stosowana przez klienty strumieni nazwanych i jest nadzbiorem flagi RPC_C_IMP_LEVEL_IDENTIFY. Oprócz możliwości wykorzystania tokena klienta do celów identyfikacji i uwierzytelniania klienta, serwer może personifikować klienta w celu uzyskania dostępu do zasobów lokalnych.

Gdy klient ma już klasę pośredniczącą i skonfigurowane odpowiednie uwierzytelnianie, może rozpocząć pracę i wywoływać metody obiektów COM. Dostawca usług zabezpieczeń (SSP) może zapewnić ochronę danych przekazywanych pomiędzy klientem i serwerem. Także w tym przypadku stosowanie wyłącznie wartości domyślnych nie pozwoli spać spokojnie.

Ochrona danych przesyłanych w sieci zależna jest od poziomu uwierzytelnienia. Możliwe jest zapewnienie integralności i poufności danych. Chociaż istnieje kilka poziomów uwierzytelniania, tylko dwa są przydatne przy defensywnym programowaniu zabezpieczeń — RPC_C_AUTHN_LEVEL_PKT_INTEGRITY oraz RPC_C_AUTHN_LEVEL_PKT_PRIVACY. Druga możliwość zapewnia pełną ochronę, obejmującą wykrywanie fałszerstw i szyfrowanie wszystkich danych, włącznie z nagłówkami protokołów. Opcja RPC_C_AUTHN_LEVEL_PKT_INTEGRITY — dzięki podpisywaniu danych kluczem sesji — zapewnia ten sam poziom wykrywania fałszerstw, jednak możliwe jest podsłuchiwanie danych. Jeśli dopuszczalny jest spadek wydajności, związany z szyfrowaniem wszystkich przekazywanych danych, należy wybierać pełną poufność pakietów. Może okazać się, że szyfrowanie nie będzie miało znaczącego wpływu na wydajność, a na pewno znacznie utrudni hakerom atak na oprogramowanie.
Omawiając uwierzytelnianie COM wspomniałem, że NTML nie pozwala na uwierzytelnienie serwera. W Windows 2000 wprowadzono protokół uwierzytelniania Kerberos — dosyć nowoczesny i zaawansowany w porównaniu z NTLM. Jedną z wyróżniających go cech jest obsługa wzajemnego uwierzytelniania i możliwość weryfikacji tożsamości serwera. Do stosowania protokołu Kerberos potrzebny jest nie tylko system Windows 2000 lub nowszy — komputery zaangażowane w komunikację muszą być członkami domeny Kerberos, która (dla odróżnienia od całkowicie odmiennego modelu domeny Windows NT®) nazywana jest domeną Windows 2000. Zakładając, że spełnione są te wymagania (a planety znalazły się we właściwym położeniu), zażądanie uwierzytelnienia Kerberos z obiektu COM przy użyciu struktury COAUTHINFO jest dosyć proste. Chcąc korzystać z uwierzytelniania Kerberos należy jednak pamiętać o tym, że tożsamość serwera COM musi posiadać uprawnienia do komunikacji sieciowej. Jeśli na przykład serwer COM zostanie uruchomiony z konta *usługa lokalna*, to tożsamość serwera nie będzie miała żadnych poświadczeń niezbędnych do przeprowadzenia uwierzytelnienia.

Ochrona danych na serwerze

Często aplikacja serwerowa przechowuje ważne informacje konfiguracyjne, takie jak łańcuchy połączeń z bazami danych lub nawet zbuforowane uwierzytelnienia użytkowników. Dane te muszą być przechowywane w bezpieczny sposób i do najlepszych praktyk należy zabezpieczenie danych na dysku lub w rejestrze za pomocą silnych list ACL. Lista ACL jest związana z plikiem lub kluczem rejestru za pośrednictwem deskryptora zabezpieczeń. Deskryptory zabezpieczeń przechowują informacje o zabezpieczeniach związane z danym obiektem. Na deskryptorach oparty jest zorientowany obiektowo model zabezpieczeń, będący najważniejszym elementem programowania zabezpieczeń dla Windows.

Deskryptor zabezpieczeń przechowuje dwie ważne informacje — identyfikator zabezpieczeń (SID), który wskazuje właściciela obiektu oraz dyskrecjonalną listę kontroli dostępu (DACL) zawierającą osoby i grupy, którym przydzielono lub odmówiono prawa dostępu do obiektu oraz wszystkie prawa przydzielone tym osobom i grupom.

Utworzenie deskryptora zabezpieczeń od podstaw związane jest z pisaniem ogromnej ilości kodu. Aby utworzyć listę DACL, należy za pośrednictwem funkcji InitializeAcl utworzyć pustą listę ACL, a następnie dla każdego identyfikatora SID, dla którego chcemy określić jakieś prawa, wywołać funkcje AddAccessAllowedAceEx oraz AddAccessDeniedAceEx. Oczywiście przed określeniem praw trzeba pobrać lub utworzyć identyfikatory SID dla różnych użytkowników i grup. Kolejną czynnością jest utworzenie rzeczywistego deskryptora zabezpieczeń poprzez wywołanie InitializeSecurityDescriptor oraz ustawienie utworzonej wcześniej listy ACL jako listy DACL deskryptora zabezpieczeń poprzez wywołanie funkcji SetSecurityDescriptorDacl. W procesie tym łatwo można popełnić błąd. Rozwiązaniem mogłoby być utworzenie funkcji pomocniczych i klas opakowujących, upraszczających ten proces. Jednak lepszym rozwiązaniem jest wykorzystanie funkcji pomocniczej wprowadzonej w Windows 2000, tworzącej deskryptor zabezpieczeń na podstawie łańcucha znaków. Funkcja ConvertStringSecurityDescriptorToSecurityDescriptor przyjmuje na wejściu specjalnie sformatowany łańcuch i tworzy pełny deskryptor zabezpieczeń. Oto przykład:

```
LocalMemory<PSECURITY_DESCRIPTOR> pSecurityDescriptor;
const std::wstring definition = L"O:BAD:(A;;FA;;;BA)(A;;FR;;;NS)";
CheckError(::ConvertStringSecurityDescriptorToSecurityDescriptor(
    definition.c_str(),SDDL_REVISION_1, &pSecurityDescriptor.m_ptr, 0));
```

Szablon klasy LocalMemory w swoim destruktorze automatycznie zwalnia posiadaną pamięć przy użyciu funkcji LocalFree. Programując zabezpieczenia często stosuję klasę LocalMemory, ponieważ wiele funkcji zabezpieczeń alokuje pamięć przy użyciu funkcji LocalAlloc. Klasa LocalMemory jest dostępna wraz z przykładowym kodem dołączonym do tego artykułu. Tokeny O: i D: wskazują sekcje właściciela i DACL w łańcuchu definicji deskryptora zabezpieczeń. Właściciel jest sklasyfikowany jako BA, co oznacza wbudowaną grupę *administratorzy*. Lista DACL opisana jest przez segment (A;;FA;;;BA)(A;;FR;;;NS), który reprezentuje dwa wpisy kontroli dostępu (ACE). Litera A w obydwu wpisach oznacza, że wpisy te służą do nadania praw dostępu. Odmówienie dostępu byłoby oznaczone literą D. FA i FR sygnalizują przyznane prawa — FA to FILE_ALL_ACCESS, a FR to FILE_GENERIC_READ. BA i NS wskazują identyfikatory SID, którym udzielane są

prawa dostępu. Jak już napisałem wcześniej, BA oznacza grupę administratorów; z kolei NS oznacza konto *usługa sieciowa*.

Szczegółowy opis formatu łańcucha znajduje się w dokumentacji w bibliotece MSDN® Library, pod hasłem Security Descriptor String Format.

Mając już podstawowe informacje o tworzeniu deskryptora zabezpieczeń możemy przejść do zabezpieczenia nowo utworzonego pliku. Tworząc nowy plik przy użyciu funkcji CreateFile, można podać także deskryptor zabezpieczeń. Deskryptor zabezpieczeń podaje się w strukturze SECURITY_ATTRIBUTES, przekazywanej do funkcji CreateFile. Jednak jeśli plik już istnieje, to deskryptor zabezpieczeń zostanie zignorowany. Wprowadza to poważne ryzyko, ponieważ atakujący może utworzyć plik ze słabymi zabezpieczeniami ACL, zanim aplikacja utworzy plik. Jeśli tworzony kod nie zostanie zabezpieczony przed atakiem tego typu, ważne dane nie będą chronione i narażone na nieautoryzowany dostęp. Aby uniknąć tych zagrożeń, należy po wywołaniu funkcji CreateFile sprawdzić wynik funkcji GetLastError. Jeśli zostanie zwrócony kod błędu ERROR_ALREADY_EXISTS, będziemy wiedzieli, że plik już istniał i będziemy mogli ponownie zastosować deskryptor bezpieczeństwa. Operację tę przedstawiono na **ilustracji 8**.

Do uaktualniania list DACL obiektu i pliku jest stosowana funkcja SetSecurityInfo. W kodzie przykładu można zauważyć, że zażądałem kilku dodatkowych praw dostępu. Zrobiłem to, aby umożliwić uaktualnienie deskryptora zabezpieczeń z wykorzystaniem uchwytu pliku, zwróconego przez funkcję CreateFile. Te prawa dostępu wykorzystywane są wyłącznie wtedy, gdy deskryptor zabezpieczeń pliku musi zostać uaktualniony. Żądane uprawnienia zawsze należy ograniczać do niezbędnego minimum. Lepszym rozwiązaniem, pozwalającym uniknąć żądania przyznania wszystkich tych uprawnień, jest zastosowanie funkcji SetNamedSecurityInfo zamiast funkcji SetSecurityInfo. Zamiast podawać uchwyt pliku będziemy wtedy mogli podać pełną ścieżkę do pliku w postaci łańcucha znaków. Funkcja ta wewnętrznie otwiera plik z odpowiednimi prawami dostępu, uaktualnia deskryptor zabezpieczeń pliku i zamyka plik.

Ochrona serwerów strumieni nazwanych

Omówiona w poprzedniej sekcji technika zabezpieczania plików może być także stosowana do zabezpieczania serwerów strumieni nazwanych oraz serwerów innych typów. Serwery strumieni nazwanych zazwyczaj implementowane są jako usługi systemu Windows. Główny wątek nasłuchujący usługi wywołuje funkcję CreateNamedPipe, aby utworzyć instancję strumienia nazwanego. Następnie wywołuje funkcję ConnectNamedPipe, która będzie czekała na połączenie z procesu klienta. Gdy klient podłączy się do strumienia, serwer może przekazać połączenie wątkowi roboczemu i utworzyć nową instancję strumienia, która będzie oczekiwać na kolejnego klienta. Oto podstawowy sposób stosowania funkcji CreateNamedPipe:

```
CHandle handle(::CreateNamedPipe(L"\\\\.\\pipe\\Kerr.SecuritySample",
    PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
    PIPE_TYPE_BYTE, PIPE_UNLIMITED_INSTANCES,
    0 /*domyślny rozmiar bufora wyjściowego */,
    0 /*domyślny rozmiar bufora wejściowgo */,
    0 /*domyślny czas wygaśniecia ważności żądania */,
    0 /*brak atrybutów zabezpieczeń */));
```

Powyższy kod utworzy nową instancję strumienia Kerr.SecuritySample o komunikacji dwukierunkowej i asynchronicznym wejściu i wyjściu danych po stronie serwera. Jakie są prawa dostępu do tego strumienia? Dla zwróconego uchwytu strumienia możemy wywołać funkcję GetSecurityInfo, a uzyskany deskryptor zabezpieczeń strumienia przekazać do funkcji ConvertSecurityDescriptorToStringSecurityDescriptor i uzyskać deskryptor zabezpieczeń w formacie łańcucha znaków. Jeśli zastosowane zostały flagi OWNER_SECURITY_INFORMATION oraz DACL_SECURITY_INFORMATION, to powinniśmy otrzymać wynik podobny do zamieszczonego poniżej:

```
O:NSD:(A;;FA;;;SY)(A;;FA;;;BA)(A;;FA;;;NS)(A;;FR;;;WD)(A;;FR;;;AN)
```

Na **ilustracji 9**. opisano znaczenie poszczególnych części łańcucha. Najwyraźniej nie jest to zbyt silna lista ACL — w liście ACL nie powinno się w ogóle stosować aliasów *Wszyscy* i *Anonimowy*. Lepszym rozwiązaniem jest jawne dostarczenie deskryptora zabezpieczeń. Deskryptor zabezpieczeń jest określany przez strukturę SECURITY_ATTRIBUTES przekazywaną jako ostatni parametr funkcji CreateNamedPipe.

Inną ważną czynnością poprawiającą ochronę serwera jest sprawdzenie, że to my jesteśmy twórcami potoku. Twórcą — a tym samym właścicielem strumienia — jest kod, który pierwszy wywołał funkcję CreateNamedPipe z unikalną nazwą strumienia. Kolejne wywołania tej funkcji powodują utworzenie nowych instancji obiektu istniejącego już strumienia. Haker mógłby ustalić nazwę strumienia i utworzyć obiekt strumienia zanim zrobi to aplikacja. Będąc twórcą strumienia ma odpowiednie prawa dostępu do wszystkich instancji strumienia.

Aby uniknąć ataków tego typu, można stosować flagę trybu dostępu FILE_FLAG_FIRST_PIPE_INSTANCE. Jeśli funkcja CreateNamedPipe wykryje, że strumień nazwany już istnieje, to jej wykonanie zakończy się niepowodzeniem.
W zależności od wymagań konieczne może okazać się utworzenie dla poszczególnych klientów dodatkowych strumieni nazwanych. Dla tych strumieni można utworzyć bardziej restrykcyjny deskryptor zabezpieczeń, który będzie dawał dostęp do strumienia tylko określonym klientom. Z tego i z innych powodów warto pobrać token reprezentujący sesję logowania klienta. Funkcja ImpersonateNamedPipeClient przypisuje token klienta do bieżącego wątku. Zakładając, że podczas podłączania do strumienia klient zastosował flagę SECURITY_IMPERSONATION (omówioną w sekcji *Ochrona klientów strumieni nazwanych*), serwer będzie miał możliwość uzyskania dostępu do zasobów lokalnych w imieniu klienta. W poniższym przykładzie zamieszczono prostą klasę, umożliwiającą przypisanie tokena klienta do bieżącego wątku.

```
class SafeImpersonateNamedPipeClient {
public:
    explicit SafeImpersonateNamedPipeClient(HANDLE pipe) {
        CheckError(::ImpersonateNamedPipeClient(pipe));
    }
    ~SafeImpersonateNamedPipeClient() {
        BOOL success = ::RevertToSelf();
        ATLASSERT(success);
    }
    };
```

Jeśli klient zastosuje flagę SECURITY_IDENTIFICATION, serwer nie będzie mógł użyć tokena wątku w celu uzyskania dostępu do zasobów, ale będzie mógł uzyskać informacje na temat grup, do których należy klient, przywilejów klienta oraz innych informacji, które mogą być przydatne w czasie autoryzacji. Aby pobrać token, należy wywołać funkcję OpenThreadToken. Destruktor klasy SafeImpersonateNamedPipeClient automatycznie zajmie się przywróceniem kontekstu zabezpieczeń serwera. Za pomocą tej klasy można w bardzo prosty sposób uzyskać token klienta, jak widać w poniższym przykładzie:

```
CHandle token;
{
    SafeImpersonateNamedPipeClient impersonate(handle);
    CheckError(::OpenThreadToken(::GetCurrentThread(), TOKEN_QUERY, TRUE, &token.m_h));
}
```

Po zakończeniu obsługi klienta zostanie wywołana funkcja RevertToSelf i w zmiennej token ponownie znajdzie się kopia tokenu serwera. Często można spotkać się z błędnym przekonaniem, że dzięki personifikacji serwer może wykonywać operacje podając się za klienta. Nie jest to prawda. W terminologii zabezpieczeń Windows personifikacja oznacza po prostu przypisanie do bieżącego wątku tokena sesji logowania klienta. To, czy serwer może uzyskać dostęp do zasobów w imieniu klienta, jest zależne od ustawień uwierzytelniania zdefiniowanych przez klienta. Próba uzyskania dostępu do systemu plików przy użyciu funkcji CreateFile przy jednoczesnej personifikacji klienta nie powiedzie się, jeśli klient nie udzielił nam do tego praw.

Mając już token, można za pomocą funkcji GetTokenInformation uzyskać na jego podstawie informacje potrzebne do przeprowadzenia dowolnego uwierzytelnienia. Alternatywnie można zarządzać własnymi prywatnymi deskryptorami zabezpieczeń aplikacji i — używając funkcji AccessCheck — implementować spójny model autoryzacji.

Strumienie nazwane obsługują podpisywanie, zapewniając tym samym poziom integralności danych, nie obsługują jednak poufności. Podpisywanie danych nie jest nawet domyślnie włączone i jest sterowane przez ogólne ustawienia systemu. Z tego powodu nie powinno się wykorzystywać strumieni do przesyłania ważnych, poufnych informacji. Jeśli istnieje konieczność stosowania strumieni, należy najpierw ustalić wspólny klucz szyfrujący (przy użyciu innej formy komunikacji międzyprocesowej, która zapewnia integralność i poufność danych), a następnie dane przesyłane strumieniem szyfrować za pomocą algorytmu szyfrowania symetrycznego.

Ochrona serwerów COM

W przypadku klientów COM sugerowałem jawne określanie wymagań bezpieczeństwa i ignorowanie wartości domyślnych komputera lub procesu. Natomiast konfigurowanie ochrony serwerów lepiej jest pozostawić administratorom. Nadal trzeba być świadomym poziomu stosowanych zabezpieczeń, ale — zamiast zastępować wszystkie ustawienia własnymi — lepiej jest zainstalować serwer z domyślnymi bezpiecznymi ustawieniami. W razie potrzeby ustawienia te mogą zostać zmodyfikowane przez administratora. W tej sekcji omówię aspekty zabezpieczeń związane z klasycznymi serwerami COM oraz aplikacjami serwerowymi COM+.

Istnieją dwa odrębne punkty kontroli dostępu do serwera COM. Pierwszy jest kontrolowany przez prawa uruchomienia lub aktywacji, a drugi jest kontrolowany prawami dostępu. Klasyczne serwery COM są zwykle aktywowane przez kod, który je wywołuje. Oznacza to, że proces, w którym pracuje serwer COM, jest uruchamiany wtedy, gdy klient po raz pierwszy aktywuje obiekt COM — na przykład przy użyciu funkcji CoCreateInstanceEx. Możliwość aktywowania serwera COM przez klienta określana jest przez prawa aktywacji. Prawa te określane są na poziomie aplikacji COM i mają formę deskryptora zabezpieczeń, przechowywanego w kluczu rejestru aplikacji COM.

Administratorzy mogą zarządzać prawami uruchamiania korzystając z przystawki *Usługi składowe* (Component Services) konsoli Microsoft® Management Console. Jeśli prawa uruchamiania nie są ustawione na poziomie aplikacji, to stosowane są wartości domyślne komputera. Wartości te można także zmieniać za pomocą przystawki *Usługi składowe*.

Drugi punkt kontroli dostępu do serwera COM to moment wywołania serwera COM przez klienta za pomocą klasy pośredniczącej. Ustawienia uwierzytelniania klasy pośredniczącej przesyłane są do serwera, który następnie może ustalić, czy klient posiada prawa dostępu do obiektu. Kontrolą dostępu także można zarządzać na poziomie aplikacji COM poprzez węzeł *DCOM Config* przystawki konsoli MMC. Serwer — gdy zostanie po raz pierwszy uruchomiony — może także nadpisać prawa dostępu poprzez wywołanie funkcji CoInitializeSecurity. Jako pierwszy parametr funkcji podaje się deskryptor zabezpieczeń, zawierający listę ACL, z którą porównywane są wszystkie wywołania RPC nadchodzące do serwera.

Chociaż klient określa poziom informacji uwierzytelniających dostarczany do serwera, serwer może zażądać od klienta minimalnej jakości usługi. Jest to kontrolowane przez poziom uwierzytelniania. W sekcji *Ochrona klientów COM* omówiłem poziom uwierzytelniania stosowany przez klienta, ponieważ to właśnie klient dokonuje wyboru poziomu. Serwer może jednak zażądać minimalnego poziomu uwierzytelnienia. Możliwość ta jest wykorzystywana zwłaszcza wtedy, gdy niezbędna jest integralność i poufność danych. Należy wprowadzić poziomy integralności pakietów i poufności pakietów, kładąc większy nacisk na poufność, ponieważ zapewnia ona wykrywanie fałszerstw oraz szyfrowanie wszystkich przesyłanych danych. Poziom uwierzytelniania danej aplikacji COM także może być kontrolowany przy użyciu przystawki *Usługi składowe* lub funkcji CoInitializeSecurity.

Gdy klient pomyślnie przejdzie przez sprawdzanie praw uruchamiania i dostępu, serwer może użyć funkcji CoImpersonateClient, aby przypisać token klienta do bieżącego wątku. Po zakończeniu personifikacji należy wywołać funkcję CoRevertToSelf, aby przywrócić kontekst zabezpieczeń serwera. Obydwie funkcje są

opakowaniami dla interfejsu IServerSecurity, do którego można bezpośrednio uzyskać dostęp się przy użyciu funkcji CoGetCallContext. To, co serwer może zrobić z tokenem użytkownika, zależy od dozwolonego poziomu personifikacji, kontrolowanego przez klienta. Sposób stosowania tokena klienta jest taki sam jak w przypadku serwerów strumieni nazwanych — aby uzyskać token z wątku, stosuje się funkcję OpenThreadToken i tak dalej. Aplikacje COM+ wyłamują się z obowiązującego w Windows tradycyjnego modelu zabezpieczeń opartego na obiektach, wykorzystującego listy kontroli dostępu. Aplikacje te stosują model, w którym prawa dostępu do metod i obiektów przydzielane są na podstawie przypisanych ról. Role te nie są związane z grupami systemu Windows i są zarządzane deklaratywnie przy użyciu przystawki *Usługi składowe*. Programista definiuje role dla danej aplikacji i w czasie instalacji przypisuje im bezpieczne wartości domyślne. Administrator serwera może następnie — w miarę potrzeb — modyfikować domyślne ustawienia ról. Dobrą cechą tego modelu jest to, że zabezpieczenia są zarządzane w sposób deklaratywny — aby wykorzystać zabezpieczenia oparte na rolach, nie trzeba w kodzie stosować żadnych specjalnych funkcji. Jeśli klient spróbuje wywołać metodę, do której na podstawie przypisanej mu roli nie ma dostępu, wywołanie zostanie odrzucone zanim dotrze do obiektu COM.

W COM wprowadzono pojęcie kontekstu wywołania. Kontekst ten zawiera informacje na temat bieżącego wywołania obiektu — w szczególności na temat kodu, który bezpośrednio ten obiekt wywołał. Dostęp do kontekstu wywołania można uzyskać poprzez interfejs IServerSecurity, zwracany przez CoGetCallContext.
W COM+ rozszerzono pojęcie kontekstu wywołania, wprowadzając informacje na temat pośrednich kontekstów zabezpieczeń w łańcuchu wywołań. Takie rozwiązanie daje także możliwość programowego sprawdzenia roli, co umożliwia szczegółową kontrolę uprawnień. Funkcjonalność ta jest udostępniana przez interfejs ISecurityCallContext, który także można uzyskać za pośrednictwem funkcji CoGetCallContext. W poniższym kodzie zilustrowano sposób sprawdzenia, czy klient ma przypisaną daną rolę:

```
bool IsCallerInRole(const CComBSTR& role) {
    CComPtr<ISecurityCallContext> spCallContext;
    CheckError(::CoGetCallContext(IID_ISecurityCallContext,
        reinterpret_cast<PVOID*>(&spCallContext)));

    VARIANT_BOOL inRole = VARIANT_FALSE;
    CheckError(spCallContext->IsCallerInRole(role, &inRole));
    return VARIANT_FALSE != inRole;
}
```

Usługi COM+ są także udostępnione w środowisku .NET Framework w przestrzeni nazw System.EnterpriseServices. Jednak coraz więcej ich funkcji i usług jest zapewnianych w sposób natywny poprzez wspólne środowisko uruchomieniowe, ASP.NET oraz usługi XML Web Service.

Podsumowanie

Techniki programowania dla Windows obejmują wiele technologii, które można razem łączyć w celu tworzenia nowych i interesujących aplikacji. Wszystkie te technologie i aplikacje muszą być zabezpieczone, by użytkownikom i firmom świadczącym usługi zapewnić odpowiednią ochronę. W przypadku każdej wykorzystywanej technologii klienckiej i serwerowej należy dobrze poznać i zrozumieć model zabezpieczeń stosowany do uwierzytelniania, autoryzacji i ochrony użytkowników oraz ich danych.

Kenny Kerr projektuje i tworzy aplikacje rozproszone dla systemu Windows. Pasjonuje się językiem C++ i programowaniem zabezpieczeń. Więcej o Kenny'm można dowiedzieć się z jego technicznego blogu http://weblogs.asp.net/kennykerr lub odwiedzając jego witrynę internetową http://www.kennyandkarin.com/kenny.