

# Szyfrowanie informacji w bazie danych, czyli czego nie widać, to nie wycieknie

Paweł Goleń, Stanisław Kipiel

Celem tego tekstu jest rozważenie różnych wariantów szyfrowania danych w bazie danych. Rozważone zostaną następujące warianty:

- Szyfrowanie całej bazy danych
- Szyfrowanie istotnych kolumn w bazie danych (jeden klucz dla kolumny)
- Szyfrowanie istotnych kolumn w bazie danych (różne klucze)

Głównym celem jest porównanie zakresów zastosowania poszczególnych metod ochrony danych poprzez przedstawienie oferowanych przez każde z rozwiązań funkcji, a także ich wad. Najwięcej uwagi zostanie poświęcone rozwiązaniu Secure.Data firmy Protegrity, które jest praktyczną implementacją koncepcji szyfrowania istotnych kolumn w bazie danych.



## 1. Szyfrowanie całej bazy danych

Pierwszym z prezentowanych rozwiązań jest szyfrowanie całej bazy danych na poziomie plików. Nie jest to rozwiązanie związane z konkretną bazą danych, lecz raczej z systemem, na którym ta baza została zainstalowana. Można to obrazowo przedstawić jako zabezpieczenie plików bazy przed niepowołanym dostępem do zawartych w nich danych, w przypadku, gdy niepowołane osoby wejdą w ich posiadanie. Wbrew pozorom nie jest to całkiem nieprawdopodobne, wystarczy wspomnieć następujące sytuacje:

- backup systemu, w którym zawarte są pliki bazy danych
- dostęp administracyjny do maszyny
- awaria dysku i jego przekazanie go do serwisu (lub złomowanie starego sprzętu – patrz MSZ)
- włamanie do systemu i skopiowanie plików

To oczywiście nie wszystkie sytuacje, w których pliki bazy mogą trafić w niepożądane ręce, ale oddają sytuacje, w których przydatne jest szyfrowanie całej bazy danych na poziomie plików.

Jak działa takie rozwiązanie? Najłatwiej wyobrazić sobie konstrukcję złożoną z kilku warstw. Na dysku przechowywane są dane w określonym formacie. Z danych tych korzysta pewna aplikacja (na przykład baza danych, choć właściwie nie ma to znaczenia), która jednak nie uzyskuje bezpośredniego dostępu do danych zapisanych na dysku. Warstwą pośredniczącą jest system operacyjny, który realizuje funkcje odczytu i zapisu danych. Warstwa ta może być dodatkowo poszerzona o funkcje kryptograficzne, które pozwalają na szyfrowanie “w locie” danych zapisywanych, oraz odszyfrowywanie danych odczytywanych z dysku. W takim przypadku aplikacja nie musi być świadoma istnienia warstwy szyfrującej i nie są wymagane żadne modyfikacje w jej kodzie. Przykładowe rozwiązania tego typu to EFS w Windows 2000 albo cryptoloop (i rozwinięcia tej koncepcji) w systemie linux i pokrewnych. Pliki bazy danych w takim przypadku umieszczane mogą być albo na szyfrowanej partycji (cryptoloop), albo oznaczone jako zaszyfrowane (EFS), a dostęp do nich będzie możliwy dopiero po podmontowaniu zaszyfrowanej partycji, lub po dostarczeniu do systemu odpowiedniego certyfikatu z kluczem prywatnym. W obu przypadkach intruz musi poza plikami bazy pozyskać również klucze (certyfikaty), które wykorzystane były do ich zaszyfrowania, a to już nie koniecznie jest zadaniem prostym.

Jakie są wady takiego rozwiązania? Oczywiście wadą jest brak jakiegokolwiek ochrony danych po rozszyfrowaniu przez system. Innymi słowy dla intruza atakującego sam serwer bazy danych lub aplikację, która z niego korzysta, takie szyfrowanie nie jest przeszkodą przed uzyskaniem dostępu do danych. Dotyczy to zarówno ataków lokalnych, jak i ataków zdalnych. Dostęp do danych w żaden sposób nie jest ograniczony również dla DBA, także jeżeli intruz będzie mógł modyfikować zapytania SQL kierowane do bazy przez aplikacje (na przykład poprzez sql injection), może uzyskać dostęp do danych, do których dostępu z założenia mieć nie powinien. Należy także uwzględnić pewną degradację wydajności, która musi nastąpić z uwagi na konieczne operacje kryptograficzne. Jeśli uwzględnić rozmiar plików wykorzystywanych przez bardziej rozbudowane bazy, oraz ilość operacji odczytu i zapisu generowanych w trakcie normalnej pracy tej bazy, wprowadzane przez warstwę szyfrującą obciążenie może być nieakceptowane. W efekcie zakres stosowania tej metody jest znacznie ograniczony. Przemawia za tym zarówno dość niski poziom oferowanego bezpieczeństwa, jak i problemy związane z wydajnością. Rozwiązanie takie stosować można więc głównie w systemach o ograniczonym obciążeniu, które nie wymagają najwyższego poziomu bezpieczeństwa oraz wysokiej wydajności.

- Zalety
  - Łatwość wdrożenia
  - Przejrzystość dla bazy danych i aplikacji
  - Ochrona danych zapisanych na dysku

- Wady
  - Brak ochrony danych po uruchomieniu serwera
  - Wprowadzenie dodatkowego obciążenia

Lepsze rozwiązanie powinno oferować ochronę danych po uruchomieniu serwera, a także zmniejszać obciążenie wprowadzane do systemu.

## 2. Szyfrowanie kolumn

Baza danych służy do przechowywania i przetwarzania informacji. Informacja przechowywana w bazie podzielona jest między tabelami, w których znajdują się rekordy. Każdy z rekordów zawiera wiele pól o różnej zawartości. Znaczna część tych informacji jest powszechnie dostępna i trudno uważać ją za poufną. Co więcej szyfrowanie informacji zawartych w każdym rekordzie w poważny sposób ograniczyłoby przydatność takiej bazy. Przykładowo rozważmy hipotetyczny sklep internetowy. Można z dużym prawdopodobieństwem założyć, że w wykorzystywanej przez sklep bazie danych będzie tabela, w której przechowywane będą dane użytkowników. W niej z kolei prawdopodobnie znajdą się następujące informacje:

- identyfikator
- login do serwisu
- dane osobowe (imię i nazwisko, adres zamieszkania)
- adres e-mail
- hasło do serwisu
- dodatkowe notatki (znaczenie tego pola stanie się jasne później)

Być może będzie również druga tabela, w której przechowywane będą informacje o aktywności użytkowników. Trudno zaprzeczyć, że informacje na temat ulubionych artykułów czy wydawanych sum pieniędzy mogą być interesujące z marketingowego lub czysto statystycznego punktu widzenia. Załóżmy więc, że ta druga tabela zawiera pola typu:

- identyfikator użytkownika
- identyfikator przedmiotu
- kwota
- data zakupu

Co się stanie, gdy ktoś w jakiś sposób wejdzie w posiadanie takiej bazy danych? Zdobędzie wiele informacji, które można uznać za poufne. Los sklepu, z którego nastąpi wyciek takich informacji będzie raczej niewesoły, z pewnością narazi się na utratę zaufania klientów, a być może również na konieczność wypłaty odszkodowań. Zrozumiałe jest więc to, że przed takimi zdarzeniami należy się chronić. Jak to zrobić? Oczywiście, można dbać, by wykorzystywana aplikacja nie miała błędów, tak by atak zdalny był bardzo mało prawdopodobny i zaakceptować takie ryzyko. Jest jednak wiele zagrożeń innego typu, choćby wcześniej wspomniane kopie zapasowe, dostęp administracyjny do maszyny (i/lub do samej bazy), czy wreszcie konieczność wysłania sprzętu do serwisu. Oczywiście, takie sytuacje mogą zostać rozwiązane proceduralnie, ale nie ma gwarancji, że procedury nie zostaną złamane, dlatego przydatne staje się kolejna warstwa zabezpieczeń, a warstwą tą może być szyfrowanie istotnych danych w bazie.

Jak przenieść to na realia rozważanego przykładu? W tabeli z danymi użytkownika informacjami, które można uznać za poufne są dane zawarte w wszystkich kolumnach, poza identyfikatorem oraz loginem do serwisu, w tabeli z informacjami o aktywności użytkownika poufna jest kwota i data zakupu. Można uznać również, że nie jest wskazane, by osoby niepowołane mogły poznać identyfikator przedmiotu, jednakże łatwo wyobrazić sobie zapytania, w których to pole będzie wykorzystywane, więc jego szyfrowanie może wprowadzić poważne problemy wydajnościowe, które raczej nie są uzasadnione wagą danych znajdujących się w tej kolumnie. Jakie informacje zdobędzie

ktoś, kto wejdzie w posiadanie takiej bazy? Zdobędzie identyfikator i login użytkownika, będzie wiedział również, że użytkownik o danym loginie zakupił przedmioty o określonych identyfikatorach, jeśli wejdzie w posiadanie całej bazy, będzie mógł nawet ustalić, o jakie przedmioty chodzi, natomiast ilość danych dotyczących tożsamości użytkownika będzie znikoma i ograniczać będzie się jedynie do jego loginu. Pozostałe dane będą zaszyfrowane i bez zdobycia klucza ich przydatność będzie znikoma.

Można zastanowić się, czy dane zawarte w loginie są rzeczywiście takie mało istotne i nie pozwalają na identyfikację użytkownika. Cóż, wielu użytkowników wybierze sobie login łatwy do zapamiętania, który zawierać będzie fragment imienia lub nazwiska. Te informacje mogą doprowadzić do odkrycia tożsamości osoby korzystającej z określonego loginu. Czy można te dane szyfrować? Oczywiście, że tak, ale należy zwrócić uwagę na fakt, że będzie to wiązało się w praktyce z wielokrotnym rozszyfrowaniem tego pola dla wszystkich rekordów znajdujących się w tabeli. W trakcie logowania do serwisu użytkownik podaje swój login oraz hasło, w celu sprawdzenia poprawności hasła konieczne jest znalezienie odpowiedniego loginu i porównanie zapisanego hasła. Czy nie można w jakiś sposób ograniczyć zakresu rekordów, na którym będzie miała miejsce operacja odszyfrowania tej kolumny? Jednym z możliwych rozwiązań jest wprowadzenie do bazy dodatkowej kolumny, która zawiera wynik funkcji skrótu (np. MD5, SHA1) wykonanej na loginie użytkownika. Wówczas weryfikacja logowania użytkownika składać będzie się z dwóch etapów, pierwszy będzie wyszukiwał te rekordy, dla których skrót podanego przez użytkownika loginu zgadza się ze skrótem przechowywanym w bazie, a następnie sprawdzać będzie, czy zgadza się również rozszyfrowany login. Po dokładniejszej analizie może okazać się również, że drugi etap jest zbędny i wystarczy samo porównanie hasha oraz hasła, ponieważ funkcja skrótu na przestrzeni możliwych nazw użytkowników zawsze generować będzie unikalny hash.

Do tej pory koncepcja ochrony informacji zawartych w bazie jest dość przejrzysta i oczywista, problem pojawia się dopiero przy wprowadzaniu jej w życie. Zagadnienia, które należy przy tym rozważyć obejmują między innymi kwestie związane z generowaniem klucza, jego bezpiecznym przechowywaniem oraz udostępnianiem określonym użytkownikom.

## Szyfrowanie przy użyciu jednego klucza

W tym przypadku zakładamy, że wszystkie informacje w jednej kolumnie zaszyfrowane są tym samym kluczem, oczywiście różne kolumny szyfrowane mogą być przy pomocy różnych kluczy. Jak generować, przechowywać i kontrolować dostęp do kluczy? Temat generowania dobrych kluczy był poruszany wielokrotnie, w związku z czym w tym miejscu nie ma sensu rozważać go po raz kolejny. Bardziej istotnymi zagadnieniami są kwestie przechowywania i kontroli dostępu do kluczy. Ponieważ sposobów realizacji tych zagadnień może być wiele, przedstawione zostaną tutaj rozwiązania wykorzystywane przez firmę Protegrity w produkcie Secure.Data.

W aplikacjach, w tym tych związanych z przetwarzaniem danych, zauważyć można coraz szersze korzystanie z pojęcia "roli". Do wykonania operacji należących do określonej roli konieczny jest dostęp do określonych zasobów (na przykład danych przechowywanych w bazie). Użytkownik otrzymuje prawa do zasobów poprzez przypisanie go do określonej roli. Koncepcja taka wykorzystana jest również przy szyfrowaniu informacji przechowywanych w bazie danych. Do poszczególnych kolumn, które mają być zaszyfrowane, przypisywane są obiekty określane jako "item". W rzeczywistości każdy z takich obiektów jest kluczem wykorzystywanym do szyfrowania danych w kolumnie, do której został przypisany. W systemie tworzone są również role, do których dodawani są następnie użytkownicy. Poszczególne role mogą mieć prawa dostępu do określonych kluczy, które dodatkowo są ograniczane do określonych operacji (select, insert, update, delete). Dzięki temu jeśli komuś uda się nawet zmodyfikować zapytanie wykonujące się w ramach roli mającej dostęp do klucza chroniącego określoną kolumnę, nie będzie on w stanie zmodyfikować jej zawartości, ponieważ z klucza tego korzystać może jedynie do operacji SELECT. Klucze przechowywane są w bazie danych i są zaszyfrowane przy pomocy klucza głównego ("master key"), który przechowywany może być w urządzeniu HSM. HSM, czyli hardware security module jest urządzeniem przeznaczonym do bezpiecznego przechowywania, generowania i zarządzania klu-

czami. Urządzenia tego typu wykonują również wszystkie operacje kryptograficzne, które wykorzystują klucze w nich składowane, dzięki czemu klucz nigdy nie opuszcza modułu HSM. Zakłada się, że w przypadku próby ingerencji w urządzenie (na przykład poprzez jego rozmontowanie) klucze w nim przechowywane zostaną zniszczone.

Poważną zaletą produktu oferowanego przez Protegrity jest dość łatwa integracja z istniejącymi aplikacjami i bazami danych. W przypadku, gdy dane w niektórych kolumnach danej tabeli mają zostać zaszyfrowane, nazwa tabeli jest zmieniana, a w jej miejsce tworzony jest nowy widok, oraz dodatkowe triggerzy przeznaczone do obsługi chronionych danych. Aplikacja korzystająca z takiego widoku nie musi być świadoma akcji, które są wykonywane w trakcie obsługi jej zapytania. Tworzenie triggerów i widoków oraz zmiana nazw tabeli wykonywana jest poprzez skrypt generowany za pomocą graficznych narzędzi wchodzących w skład Secure.Data. Skrypt taki należy następnie uruchomić w bazie danych. Po uruchomieniu generuje on log z wykonywanych akcji, dzięki czemu w przypadku niepowodzenia można ustalić przyczynę i rozwiązać problem.

Takie podejście zakładające prostą integrację z istniejącymi rozwiązaniami ma oczywiście swoje wady. W szczególności zwracana jest uwaga na brak dodatkowego uwierzytelnienia przed operacjami z wykorzystaniem klucza. Oznacza to, że duża odpowiedzialność spoczywa na mechanizmach samej bazy danych odpowiadających za uwierzytelnianie użytkowników. Jeśli intruz będzie mógł w jakiś sposób podszyć się pod innego użytkownika, wówczas bez dodatkowych utrudnień będzie on w stanie uzyskać dostęp do danych dostępnych dla ról, do których ów użytkownik należy. Przykładowo można rozważyć sytuację, w której nieuczciwy DBA chce poznać pewne dane przechowywane w bazie. Aby chronić się przed tym istotne kolumny zostały zaszyfrowane i dostęp do nich mają jedynie role, do których nie należy DBA. Jeśli dodatkowo DBA nie ma praw do zarządzania rolami, w tym do przypisywania do nich użytkowników, rozwiązanie wydaje się być skuteczne. Niestety, DBA jest w stanie zmienić hasło użytkownika należącego do określonej roli, korzystając z jego uprawnień pozyskać interesujące go informacje, a następnie przywrócić oryginalne hasło użytkownika. Rozwiązaniem dla tego konkretnego problemu może być przeniesienie uwierzytelniania użytkowników poza serwer bazy danych, na przykład do zewnętrznego serwera LDAP.

- Zalety
  - Szyfrowanie istotnych informacji wewnątrz bazy danych
  - Mechanizmy zarządzania kluczami oraz kontroli dostępu do nich
  - Łatwość integracji rozwiązania w istniejące aplikacje
  - Zmniejszenie dodatkowego narzutu na szyfrowanie do niezbędnego minimum
- Wady
  - Brak dodatkowego uwierzytelnienia użytkownika przed przyznaniem dostępu do klucza
  - Możliwość obejścia ochrony poprzez podszywanie się pod innego użytkownika

Trzeba pamiętać, że rozwiązanie firmy Protegrity nie jest jedynym istniejącym na rynku. Do szyfrowania danych na poziomie kolumn wykorzystać można na przykład istniejący w Oracle pakiet DBMS\_OBFUSCATION\_TOOLKIT. Co oferuje DBMS\_OBFUSCATION\_TOOLKIT? Pozwala on na szyfrowanie oraz odszyfrowywanie danych z wykorzystaniem algorytmu DES, a także udostępnia możliwość korzystania z algorytmu MD5 służącego do wyliczania skrótów kryptograficznych z danych, które następnie mogą być wykorzystywane do weryfikacji integralności danych. Pakiet ten eliminuje również słabości istniejące w DBMS\_RANDOM, które prowadzić mogą do generowania słabych, przewidywalnych kluczy. Zalecane jest, by przy generowaniu kluczy wykorzystać procedurę GetKey z DBMS\_OBFUSCATION\_TOOLKIT, nie funkcje dostępne w pakiecie DBMS\_RANDOM. Procedura GetKey spełnia wymogi FIPS-140 stawiane generatorom liczb pseudolosowych. DBMS\_OBFUSCATION\_TOOLKIT stanowi więc solidną podstawę pod implementację własnych rozwiązań, nie oferuje jednak mechanizmów zarządzania kluczami, od których w znacznym stopniu zależy bezpieczeństwo całego rozwiązania.

## Szyfrowanie przy użyciu wielu różnych kluczy

Szyfrowanie przy użyciu jednego klucza ma jedną wadę. Jeśli ktoś wejdzie w jego posiadanie, lub będzie mógł modyfikować zapytanie, które do tego klucza ma dostęp, będzie mógł odczytać wszystkie dane w tej kolumnie. W zasadzie “to tak ma działać”, ale można pokusić się o dodatkowe rozważania, czy nie można tego zrealizować inaczej. Właśnie po to w rozważanym przykładzie w danych użytkownika dodane zostało pole dodatkowe notatki. W tych rozważaniach założmy, że znajdują się w polu tym znajdują się takie dane, do których powinien mieć dostęp tylko użytkownik, który jest opisywany przez dany rekord. Ponieważ ciężko jest ochronę taką osiągnąć przy pomocy wbudowanych w bazy mechanizmów, może i w tym przypadku pomocne będą rozwiązania wykorzystujące kryptografię. Można wyobrazić sobie dodatkową tabelę, w której przechowywane są klucze unikalne dla poszczególnych użytkowników. Oczywiście nie są one przechowywane w formie jawnej, lecz są zaszyfrowane, a do ich odszyfrowania potrzebne są informacje znane tylko jednej, konkretnej osobie, czyli odpowiedniemu użytkownikowi. Do zaszyfrowania klucza może być wykorzystane przykładowo dodatkowe hasło, które nie jest przechowywane nigdzie w bazie danych, lub w bardziej złożonych scenariuszach – certyfikat klienta.

## 3. Rozwiązanie dedykowane

Rozwiązanie dedykowane do jednego, konkretnego zastosowania zwykle jest lepsze, niż rozwiązanie ogólne, do wielu zastosowań. W przypadku rozwiązań ogólnych w wielu przypadkach można znaleźć sytuacje, w których pewne rzeczy nie są (łatwo) realizowalne, a w pewnej konkretnej sytuacji są wymagane. Może to oznaczać, że konieczne jest stworzenie rozwiązania uwzględniającego specyficzne wymagania. Szczególnie ma to sens w przypadku, gdy tworzony jest system od podstaw. W takiej sytuacji przykładowo można założyć, że część informacji w bazie przechowywana jest jedynie w postaci zaszyfrowanej, natomiast ich przetworzenie do użytecznej formy wykonywane jest już w samej aplikacji pracującej na tych danych.

Jako bazę dla rozwiązań dedykowanych można wykorzystać dostępne rozwiązania ogólne (na przykład Secure.Data), albo stworzyć rozwiązanie opierające się na pakiecie DBMS\_OBFUSCATION\_TOOLKIT obudowując go w bardziej wyszukane mechanizmy zarządzania kluczami, lub inne funkcje wymagane w konkretnym systemie.