Twoja lektura obowiązkowa!

Łamanie i zabezpieczanie aplikacji w systemie iOS

HELION

O'REILLY®

Jonathan Zdziarski

Tytuł oryginału: Hacking and Securing iOS Applications

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-5147-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/lamzab
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 10.

Implementowanie algorytmów szyfrowania

Szyfrowanie to jedna z najskuteczniejszych form ochrony danych aplikacji, a więc należy je wyjątkowo starannie zaimplementować. Niestety, jak przekonałeś się w poprzednich rozdziałach, nie jest to wcale łatwe. Istnieje wiele publikacji na ten temat, ale są one często zawiłe i niezrozumiałe, a wachlarz dostępnych metod i algorytmów jest naprawdę bardzo szeroki. Dlatego właśnie większość hakerów skupia się na samej implementacji algorytmu szyfrowania, zamiast próbować własnoręcznie rozszyfrować dane. Danych nie można skutecznie chronić, jeśli po rozszyfrowaniu zostaną zapisane w pamięci. Dlatego najważniejsze w implementacji szyfrowania jest niedopuszczanie do takich sytuacji. Nieużywane dane muszą być tak zabezpieczone, aby w razie kradzieży lub sklonowania urządzenia nie dało się ich rozszyfrować, przynajmniej nie bez użycia klastrów potężnych komputerów. W tym rozdziale znajduje się opis różnych technik szyfrowania i wymieniania kluczy, które znacznie utrudniają przeprowadzenie ataku na aplikację.

Siła hasła

Skuteczność wszystkich dobrych algorytmów szyfrowania zależy od siły klucza, który w większości aplikacji jest chroniony hasłem. Można zatem powiedzieć, że bezpieczeństwo takich implementacji algorytmów szyfrujących zależy przede wszystkim od siły hasła ustanowionego przez użytkownika.

Choćbyś napisał najdoskonalszą implementację, cały Twój wysiłek pójdzie na marne, jeżeli w Twoim programie będzie możliwość używania słabych haseł. Dlatego też obok mechanizmu szyfrującego drugim filarem bezpieczeństwa aplikacji są zasady dotyczące tworzenia haseł. Wymuszając stosowanie haseł o odpowiedniej długości i właściwym stopniu skomplikowania, można sprawić, że przeprowadzenie udanego ataku na program stanie się prawie niemożliwe.

Silne hasło powinno mieć następujące cechy:

- duża liczba znaków;
- połaczenie małych i dużych liter;
- zawartość cyfr;
- obecność specjalnych znaków, takich jak # i znaki interpunkcyjne;

- brak typowych wzorców klawiszowych, jak np. ciąg QWERTY;
- brak słów należących do słowników popularnych języków;
- brak dat i innych tego typu danych.

Przestrzeganie niektórych z tych zasad, np. długości hasła, można bardzo łatwo wyegzekwować. Za bezpieczne aktualnie uznaje się hasła o długości przynajmniej 12 znaków. Nietrudno też zmusić użytkownika do stosowania określonych mieszanek znaków, a dobre narzędzie do weryfikacji haseł może nawet wykrywać odległość między naciśniętymi klawiszami, aby wykryć ewentualne wzorce.

Prosty mechanizm sprawdzania haseł można zaimplementować, stosując system punktowy polegający na przyznawaniu hasłu od jednego do trzech punktów za to, ile razy każda z określonych cech występuje w tym haśle. Na przykład hasło zawierające cyfry może otrzymać maksymalnie trzy punkty, jeśli będzie zawierało przynajmniej trzy cyfry itd. Przedstawiony na listingu 10.1 program sprawdza długość hasła oraz czy zawiera ono cyfry, małe i wielkie litery oraz znaki specjalne, a także sprawdza odległość od siebie poszczególnych klawiszy i przyznaje dodatkowe punkty, jeśli nie da się w nich wykryć żadnego wzorca.

Listing 10.1. Narzędzie do sprawdzania siły haseł (passphrase_strength.m)

```
#include <stdio.h>
#include <string.h>
#include <sys/param.h>
#include <ctype.h>
#include <stdlib.h>

int key_distance(char a, char b) {
    const char *qwerty_lc = "`1234567890-="
                            "qwertyuiop[]\\"
                            " asdfghjkl;' "
                            "  zxcvbnm,./ ";
    const char *qwerty_uc = "~!@#$%^&*()_+"
                            "QWERTYUIOP{}|"
                            " ASDFGHJKL:\" "
                            "  ZXCVBNM<>? ";
    int pos_a, pos_b, dist;

    if (strchr(qwerty_lc, a))
        pos_a = strchr(qwerty_lc, a) - qwerty_lc;
    else if (strchr(qwerty_uc, a))
        pos_a = strchr(qwerty_uc, a) - qwerty_uc;
    else
        return -2;

    if (strchr(qwerty_lc, b))
        pos_b = strchr(qwerty_lc, b) - qwerty_lc;
    else if (strchr(qwerty_uc, b))
        pos_b = strchr(qwerty_uc, b) - qwerty_uc;
    else
        return -1;

    dist = abs((pos_a/13) - (pos_b/13))
         + abs(pos_a % 13 - pos_b % 13);
    return dist;
}

int score_passphrase(const char *passphrase) {
    int total_score = 0;
    int unit_score;
```

```
    int distances[strlen(passphrase)];
    int i;

    /* Długość hasła */
    unit_score = strlen(passphrase) / 4;
    total_score += MIN(3, unit_score);

    /* Wielkie litery */
    for(unit_score = i = 0; passphrase[i]; ++i)
        if (isupper(passphrase[i]))
            unit_score++;
    total_score += MIN(3, unit_score);


    /* Małe litery */
    for(unit_score = i = 0; passphrase[i]; ++i)
        if (islower(passphrase[i]))
            unit_score++;
    total_score += MIN(3, unit_score);

    /* Cyfry */
    for(unit_score = i = 0; passphrase[i]; ++i)
        if (isdigit(passphrase[i]))
            unit_score++;
    total_score += MIN(3, unit_score);


    /* Znaki specjalne */
    for(unit_score = i = 0; passphrase[i]; ++i)
        if (!isalnum(passphrase[i]))
            unit_score++;
    total_score += MIN(3, unit_score);


    /* Odległości między klawiszami */
    distances[0] = 0;
    for(unit_score = i = 0; passphrase[i]; ++i) {
        if (passphrase[i+1]) {
            int dist = key_distance(passphrase[i], passphrase[i+1]);
            if (dist > 1) {
                int j, exists = 0;
                for(j=0;distances[j];++j)
                    if (distances[j] == dist)
                        exists = 1;
                if (!exists) {
                    distances[j] = dist;
                    distances[j+1] = 0;
                    unit_score++;
                }
            }
        }
    }
    total_score += MIN(3, unit_score);

    return ((total_score / 18.0) * 100);
}


int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Składnia: %s <hasło>\n", argv[0]);
        return EXIT_FAILURE;
    }
    printf("Siła hasła: %d%%\n", score_passphrase(argv[1]));
    return EXIT_SUCCESS;
}
```

Funkcja `score_passphrase` zwraca wartość procentową od 0 do 100, określającą jakość hasła. Do programu można by jeszcze dodać moduł sprawdzający na podstawie słownika, czy w haśle nie ma jakichś słownikowych haseł. To jednak stanowi dodatkowe obciążenie dla procesora i może powodować opóźnienia w tworzeniu nowych haseł.

Uwaga na generatory losowych haseł

Generatory losowych haseł bywają bardzo pomocne w tworzeniu wysokiej jakości haseł, ale czasami mogą ułatwiać dokonywanie ataków metodą brutalnej siły. Algorytmy stosujące łatwe do wpisania wzorce ograniczają liczby możliwych kombinacji. Przeprowadzenie udanego ataku na taką aplikację może być nawet łatwiejsze niż na aplikację niemającą żadnej kontroli jakości haseł, ponieważ atakujący wie, że wszystkie hasła są tworzone według określonego wzorca.

Wprowadzenie do Common Crypto

Biblioteka Common Crypto, zwana też CCCrypt i 3CC, zawiera kilka rodzajów algorytmów szyfrujących. Obsługuje m.in. takie standardy jak AES, DES i 3DES. W zależności od używanego algorytmu można stosować szyfry blokowe i strumieniowe.

Szyfr blokowy dzieli dane na bloki o jednakowym rozmiarze i szyfruje każdy z nich z osobna, a następnie składa te bloki z powrotem w jedną całość. Tego typu szyfrowanie stosuje się w algorytmach z kluczem prywatnym. Technika ta jest bardzo wydajna, gdy rozmiar danych wejściowych jest z góry znany. Natomiast **szyfr strumieniowy** jest stosowany do szyfrowania danych o dużych rozmiarach lub strumieniowych, których nie da się zaszyfrować jako całości. Szyfry strumieniowe są zazwyczaj szybsze od blokowych, ale są również nieodporne na pewne formy ataku, takie jak przerzucanie bitów (ang. *bit flipping*) czy powtórne użycie klucza (ang. *key replay*). Szyfr strumieniowy wymaga synchronizacji, ponieważ dane są przesyłane strumieniowo do algorytmu szyfrującego.

Kolejną funkcją udostępnianą przez bibliotekę Common Crypto jest **łańcuchowanie szyfrowanych bloków**. W trybie tym każdy blok jest najpierw mieszany za pomocą operacji XOR z zaszyfrowanym tekstem z poprzedniego bloku, a dopiero potem zostaje sam zaszyfrowany. Dzięki temu każdy zaszyfrowany blok jest zależny od wszystkich poprzednich bloków. Zwiększa to poziom bezpieczeństwa, ponieważ haker, dokonując ataku typu człowiek pośrodku, nie może zmodyfikować danych w określonym miejscu w strumieniu, nie naruszając całego łańcucha od tego miejsca do końca. Uniemożliwia to także przeprowadzanie ataków polegających na ponownym wstawianiu do połączenia określonych zaszyfrowanych pakietów. Łańcuchowanie stosuje się w kombinacji z **wektorem inicjującym** (ang. *initialization vector*), czyli losową wartością używaną do zaszyfrowania pierwszego bloku. Jeśli wektor inicjujący zostanie dobrze zaimplementowany, to dla wielu kopii tych samych danych zostanie zwrócony inny tekst szyfru, co uniemożliwi przeprowadzenie ataków metodą powtórkową (ang. *replay*) i kryptoanalitycznych. Ponadto uniemożliwia to atakującemu rozszyfrowanie danych, nawet jeśli uda mu się zdobyć klucz szyfrowania, ponieważ potrzebna jest jeszcze znajomość wektora inicjującego.

Szyfry blokowe określa się jako **bezstanowe**, mimo iż w procesie łańcuchowania informacje z jednego bloku wykorzystywane są do zaszyfrowania kolejnego, ponieważ na koniec wszystkie informacje oprócz samego tekstu szyfru są usuwane. Natomiast szyfry strumieniowe zalicza się do **stanowych**, ponieważ wiadomo w procesie szyfrowania, w którym miejscu odbywa się akcja.

Operacje bezstanowe

Najprostszym sposobem użycia biblioteki Common Crypto jest bezstanowe zaszyfrowanie lub rozszyfrowanie danych. W bibliotece tej znajduje się funkcja o nazwie `CCCrypt`, której opis w dokumentacji jest następujący: „bezstanowa jednorazowa operacja szyfrowania lub deszyfrowania". Funkcja ta wykonuje wszystkie działania potrzebne do zaszyfrowania lub rozszyfrowania danych w tle. Programista musi tylko podać klucz oraz kilka parametrów i buforów.

Prototyp funkcji `CCCrypt` jest następujący:

```
CCCryptorStatus
    CCCrypt(CCOperation op, CCAlgorithm alg, CCOptions options,
    const void *key, size_t keyLength, const void *iv,
    const void *dataIn, size_t dataInLength,
    void *dataOut, size_t dataOutAvailable,
    size_t *dataOutMoved);
```

`CCOperation op`

Może być `kCCEncrypt` albo `kCCDecrypt`. Parametr ten określa, czy dane wejściowe mają zostać zaszyfrowane, czy rozszyfrowane.

`CCAlgorithm alg`

Określa, który algorytm szyfrowania ma zostać użyty. Aktualnie dostępne są następujące algorytmy: `kCCAlgorithmAES128`, `kCCAlgorithmDES`, `kCCAlgorithm3DES`, `kCCAlgorithmCAST`, `kCCAlgorithmRC4`, `kCCAlgorithmRC2` oraz `kCCAlgorithmBlowfish`.

`CCOptions options`

Określa opcje szyfrowania reprezentowane w zmiennej jako flagi. Aktualnie obsługiwane są dwie opcje: `kCCOptionPKCS7Padding` i `kCCOptionECBMode`. Pierwsza instruuje funkcję `CCCryptor`, aby w swoich operacjach przyjmowała, że stosowane jest dopełnienie `PKCS7`. Druga natomiast włącza tryb szyfrowania Electronic Code Block (ECB), w którym każdy blok jest szyfrowany osobno. Trybu ECB powinni używać tylko doświadczeni programiści, którzy dobrze wiedzą, do czego on służy, ponieważ tryb ten może powodować osłabienie bezpieczeństwa, jeśli nie zostanie właściwie zaimplementowany.

`const void *key`

`size_t keyLength`

Klucz szyfrowania i długość tego klucza. Długość klucza w dużym stopniu zależy od typu szyfrowania. Aktualnie dostępne są następujące długości kluczy: `kCCKeySizeAES128`, `kCCKeySizeAES192`, `kCCKeySizeAES256`, `kCCKeySizeDES`, `kCCKeySize3DES`, `kCCKeySizeMinCAST`, `kCCKeySizeMaxCAST`, `kCCKeySizeMinRC4`, `kCCKeySizeMaxRC4`, `kCCKeySizeMinRC2`, `kCCKeySizeMaxRC2`, `kCCKeySizeMinBlowfish` oraz `kCCKeySizeMaxBlowfish`.

`const void *iv`

Wektor inicjujący, dzięki któremu każdy szyfr jest niepowtarzalny. Dzięki jego zastosowaniu niemożliwe jest dokonywanie ataków powtórzeniowych i kryptoanalitycznych, ponieważ ten sam tekst zaszyfrowany przy użyciu tego samego klucza dzięki wektorowi inicjującemu da za każdym razem inny wynik. Do szyfrowania zawsze powinno się używać losowego wektora inicjującego.

`const void *dataIn`

`size_t dataInLength`

Dane do zaszyfrowania lub rozszyfrowania. Muszą one zostać dopełnione do rozmiaru bloku.

`void *dataOut`

`size_t dataOutAvailable`

`size_t *dataOutMoved`

Bufor wyjściowy `dataOut` alokowany przez wywołującego i przeznaczony do przechowywania czystego lub zaszyfrowanego tekstu, w zależności od rodzaju operacji. Rozmiar bufora określa się w zmiennej `dataOutAvailable`. Liczba zaszyfrowanych lub rozszyfrowanych bajtów jest zapisana w zmiennej, której adres zawiera parametr `dataOutMoved`. Rozmiar danych wyjściowych nie przekracza rozmiaru danych wejściowych plus rozmiar bloku.

Przedstawiony na listingu 10.2 program pobiera tekst z wiersza poleceń i szyfruje go przy użyciu losowo utworzonego klucza. Dzięki temu, że bibliotekę Common Crypto obsługuje zarówno system Mac OS X, jak i iOS, kod ten można skompilować w obu tych systemach.

Listing 10.2. Szyfrowanie tekstu algorytmem AES-128 (textcrypt.m)

```
#include <CommonCrypto/CommonCryptor.h>
#include <Foundation/Foundation.h>
#include <stdio.h>

int encryptText(const unsigned char *clearText) {
    CCCryptorStatus status;
    unsigned char cipherKey[kCCKeySizeAES128];
    unsigned char cipherText[strlen(clearText) + kCCBlockSizeAES128];
    size_t nEncrypted;
    int i;

    printf("Szyfrowanie tekstu: %s\n", clearText);

    printf("Użyty klucz szyfrowania: ");
    for(i=0;i<kCCKeySizeAES128;++i) {
        cipherKey[i] = arc4random() % 255;
        printf("%02x", cipherKey[i]);
    }
    printf("\n");

    status = CCCrypt(kCCEncrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        cipherKey,
        kCCKeySizeAES128,
        NULL,
        clearText, strlen(clearText),
        cipherText, sizeof(cipherText),
```

```
        &nEncrypted);
    if (status != kCCSuccess) {
        printf("Funkcja CCCrypt() zwróciła błąd %d\n", status);
        return status;
    }

    printf("Zaszyfrowano %ld bajtów\n", nEncrypted);
    for(i=0;i<nEncrypted;++i)
        printf("%02x", (unsigned int) cipherText[i]);

    printf("\n");
    return 0;
}

int main(int argc, char *argv[]) {

    if (argc < 2) {
        printf("Składnia: %s <tekst do zaszyfrowania>\n", argv[0]);
        return EXIT_FAILURE;
    }
    encryptText(argv[1]);
}
```

W systemie Mac OS X program ten można skompilować przy użyciu kompilatora `gcc`.

```
$ gcc -o textcrypt textcrypt.m -lobjc
```

Uruchom program i obserwuj, co się dzieje.

```
$ ./textcrypt "The quick brown fox jumped over the lazy dog"

Szyfrowanie tekstu: The quick brown fox jumped over the lazy dog
Użyty klucz szyfrowania: 606c64fd3adc1c684be94f5fdf1cc718

Zaszyfrowano 48 bajtów
0d462b3ec789cfafc5f0fbba49cc73507015ac24ec548bd1ef5a45a77eb34985296256a1c0073021b26c
ebc75b63aeb
```

Aby rozszyfrować tekst, należy odwrócić czynności. Program przedstawiony na listingu 10.3 dekoduje dane wejściowe do postaci surowych bajtów, a następnie wywołuje funkcję `CCCrypt`, aby rozszyfrować tekst szyfru do oryginalnej postaci.

Listing 10.3. Deszyfrowanie tekstu przy użyciu algorytmu AES-128 (textdecrypt.m)

```
#include <CommonCrypto/CommonCryptor.h>
#include <Foundation/Foundation.h>
#include <stdio.h>

int decode(unsigned char *dest, const char *buf) {
    char b[3];
    int i;

    b[2] = 0;
    for(i=0;buf[i];i+=2) {
        b[0] = buf[i];
        b[1] = buf[i+1];
        dest[i/2] = (int) strtol(b, NULL, 0x10);
    }
    return 0;
}

int decryptText(
    const unsigned char *cipherKey,
    const unsigned char *cipherText
) {
```

```
    CCCryptorStatus status;
    int len = strlen(cipherText) / 2;
    unsigned char clearText[len];
    unsigned char decodedCipherText[len];
    unsigned char decodedKey[len];
    size_t nDecrypted;
    int i;

    decode(decodedKey, cipherKey);
    decode(decodedCipherText, cipherText);
    printf("Deszyfrowanie...\n");

    status = CCCrypt(kCCDecrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        decodedKey,
        kCCKeySizeAES128,
        NULL,
        decodedCipherText, len,
        clearText, sizeof(clearText),
        &nDecrypted);
    if (status != kCCSuccess) {
        printf("Funkcja CCCrypt() zwróciła błąd %d\n", status);
        return status;
    }

    printf("Rozszyfrowano %ld bajtów\n", nDecrypted);
    printf("=> %s\n", clearText);

    return 0;
}

int main(int argc, char *argv[]) {

    if (argc < 3) {
        printf("Składnia: %s <klucz> <zaszyfrowany tekst>\n", argv[0]);
        return EXIT_FAILURE;
    }
    decryptText(argv[1], argv[2]);
}
```

Do kompilacji tego programu użyj kompilatora `gcc`.

```
$ gcc -o textdecrypt textdecrypt.m -lobjc
```

Aplikacja przyjmuje dwa argumenty: klucz szyfrowania użyty podczas szyfrowania tekstu i zaszyfrowany tekst z programu `textencrypt`.

```
$
./textdecrypt
606c64fd3adc1c684be94f5fdf1cc7180d462b3ec789cfafc50f0bba49cc73507015ac24ec548bd1ef
↪5a45a770eb
Deszyfrowanie...
Rozszyfrowano 44 bajtów
=> The quick brown fox jumped over the lazy dog
```

Szyfrowanie stanowe

Korzystanie z biblioteki Common Crypto w trybie stanowym wymaga utworzenia obiektu klasy `CCCryptor` oraz zainicjowania go kluczem i konfiguracją, które dostarcza się jako typ danych `CCCryptorRef`. Następnie wprowadza się dane wejściowe, a dane do bufora wyjściowego są wysyłane po każdym wywołaniu funkcji `CCCryptorUpdate`. W przypadku szyfru

blokowego na wejściu można podać pojedynczy blok danych (w razie potrzeby dopełniony, aby spełniać wymagania dotyczące rozmiaru bloku). W przypadku szyfru strumieniowego na wejściu można podać dane dowolnej długości, a na wyjściu otrzymuje się dane o takiej samej długości. Po zaszyfrowaniu lub rozszyfrowaniu wszystkich danych następuje opróżnienie obiektu za pomocą funkcji `CCCryptorFinal` i zapis wyniku. Następnie obiekt można zwolnić przy użyciu funkcji `CCCryptorRelease`. Obiektu klasy `CCCryptor` można wielokrotnie używać do strumieniowania lub innych operacji stanowych. Nie trzeba go inicjować dla każdego nowego pakietu. Implementacja szyfrowania stanowego jest przedstawiona na listingu 10.4. Dodany został losowy wektor inicjujący. Aby ułatwić wdrożenie tego kodu w aplikacjach, do pracy z danymi została użyta klasa `NSData`.

Listing 10.4. Funkcja szyfrująca wykorzystująca stanowy obiekt szyfrowania (stateful_crypt.m)

```
#include <CommonCrypto/CommonCryptor.h>
#include <Foundation/Foundation.h>
#include <stdio.h>

NSData *encrypt_AES128(
    NSData *clearText,
    NSData *key,
    NSData *iv
) {
    CCCryptorStatus cryptorStatus = kCCSuccess;
    CCCryptorRef cryptor = NULL;
    NSData *cipherText = nil;
    size_t len_outputBuffer = 0;
    size_t nRemaining = 0;
    size_t nEncrypted = 0;
    size_t len_clearText = 0;
    size_t nWritten = 0;
    unsigned char *ptr, *buf;
    int i;

    len_clearText = [ clearText length ];

    cryptorStatus = CCCryptorCreate( kCCEncrypt,
                                kCCAlgorithmAES128,
                                kCCOptionPKCS7Padding,
                                (const void *) [ key bytes ],
                                kCCBlockSizeAES128,
                                (const void *) [ iv bytes ],
                                &cryptor
                            );

    /* Określenie rozmiaru danych wyjściowych na podstawie rozmiaru danych wejściowych */
    len_outputBuffer = CCCryptorGetOutputLength(cryptor, len_clearText, true);
    nRemaining = len_outputBuffer;
    buf = calloc(1, len_outputBuffer);
    ptr = buf;

    cryptorStatus = CCCryptorUpdate(
        cryptor,
        (const void *) [ clearText bytes ],
        len_clearText,
        ptr,
        nRemaining,
        &nEncrypted
    );
```

```
    ptr += nEncrypted;
    nRemaining -= nEncrypted;
    nWritten += nEncrypted;

    cryptorStatus = CCCryptorFinal(
        cryptor,
        ptr,
        nRemaining,
        &nEncrypted
    );

    nWritten += nEncrypted;
    CCCryptorRelease(cryptor);

    cipherText = [ NSData dataWithBytes: (const void *) buf
                                 length: (NSUInteger) nWritten ];

    free(buf);
    return cipherText;
}
```

Aby użyć tej funkcji, należy utworzyć losowy klucz i wektor inicjujący. Następnie należy jej przekazać te informacje wraz z tekstem do zaszyfrowania. Funkcja zwraca obiekt `NSData` zawierający zaszyfrowany tekst. Na listingu 10.5 znajduje się przykład użycia powyższej funkcji. Ten zawierający funkcję `main` kod można dodać do pliku źródłowego z funkcją, aby otrzymać program szyfrujący dane w wierszu poleceń.

Listing 10.5. Przykład użycia funkcji encrypt_AES128

```
int main(int argc, char *argv[]) {
    NSData *clearText, *key, *iv, *cipherText;
    unsigned char u_key[kCCKeySizeAES128], u_iv[kCCBlockSizeAES128];
    int i;

    NSAutoreleasePool *pool = [ [ NSAutoreleasePool alloc ] init ];

    if (argc < 2) {
        printf("Składnia: %s <czystytekst>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Generowanie losowego klucza i wektora inicjującego */
    for(i=0;i<sizeof(key);++i)
        u_key[i] = arc4random() % 255;
    for(i=0;i<sizeof(iv);++i)
        u_iv[i] = arc4random() % 255;

    key = [ NSData dataWithBytes: u_key length: sizeof(key) ];
    iv  = [ NSData dataWithBytes: u_iv  length: sizeof(iv)  ];
    clearText = [ NSData dataWithBytes: argv[1] length: strlen(argv[1]) ];

    cipherText = encrypt_AES128(clearText, key, iv);

    for(i=0;i<[ cipherText length];++i)
        printf("%02x", ((unsigned char *) [ cipherText bytes ])[i]);
    printf("\n");

    [ pool release ];
}
```

Program do użytku na komputerze można skompilować za pomocą kompilatora `gcc`.

```
$ gcc -o stateful_crypt stateful_crypt.m -lobjc -framework Foundation
```

Aby móc go przetestować w urządzeniu iOS, należy go skopiować przy użyciu kompilatora krzyżowego z Xcode.

```
$ export PLATFORM=/Developer/Platforms/iPhoneOS.platform
$ $PLATFORM/Developer/usr/bin/arm-apple-darwin10-llvm-gcc-4.2 \
    -o stateful_crypt stateful_crypt.m \
    -isysroot $PLATFORM/Developer/SDKs/iPhoneOS5.0.sdk \
    -framework Foundation -lobjc
```

Szyfrowanie z kluczem głównym

W przedstawionych do tej pory przykładowych programach do szyfrowania używany był losowo wybierany klucz. Wiąże się z tym problem ochrony tego klucza. Wiemy już, że złamanie zabezpieczeń pęku kluczy urządzenia jest możliwe, a więc to rozwiązanie nie wchodzi w grę. Główny klucz szyfrowania musi być gdzieś przechowywany, ale musi też być zabezpieczony. W poprzednich rozdziałach dowiedziałeś się, że w dobrych implementacjach do rozszyfrowania danych potrzebne są informacje podawane przez użytkownika. Wówczas szyfrowanie zależy od „czegoś, co mamy" (danych i zaszyfrowanego klucza głównego) oraz „czegoś, co wiemy" (hasła). **Funkcje derywacyjne kluczy** (ang. *key derivation function* — KDF) tworzą klucze na bazie jakiejś tajnej wartości, np. hasła. Funkcja KDF przyjmuje taką wartość i wykonując serię permutacji, tworzy z niej klucz szyfrowania o żądanym rozmiarze. Tergo klucza można następnie użyć do zaszyfrowania klucza głównego.

Nasuwa się pytanie, po co w ogóle używać głównego klucza szyfrowania, zamiast po prostu użyć klucza derywowanego. Główny klucz szyfrowania, jeśli jest cały czas chroniony, nigdy nie musi być zmieniany. Gdy użytkownik zmieni swoje hasło, to wystarczy ponownie wygenerować klucz główny przy użyciu nowego klucza derywowanego. Gdyby dane były bezpośrednio powiązane z hasłem, to przy każdej zmianie hasła trzeba by było wszystkie te dane szyfrować od nowa. Inną zaletą tego podejścia jest to, że można zaszyfrować na różne sposoby wiele kopii klucza głównego. Na przykład druga kopia tego klucza mogłaby zostać zaszyfrowana przy użyciu klucza derywowanego z odpowiedzi na pytania bezpieczeństwa (np. Jak ma na imię Twój zwierzak?). Takie pytania są przydatne, gdy użytkownik zapomni swojego hasła. Ponadto aplikacja może zezwalać, aby wielu użytkowników korzystało z tych samych zaszyfrowanych danych, np. poprzez sieć albo iCloud. Szyfrując te wspólne dane jednym kluczem głównym, aplikacja może przechowywać wiele kopii wspólnego klucza głównego i chronić go kluczami derywowanymi z haseł użytkowników.

Nie wszystkie aplikacje chronią klucz główny przy użyciu funkcji derywującej klucze, przez co są bardziej podatne na niektóre rodzaje ataku. Najczęściej spotykanym niewłaściwym sposobem użycia hasła jest utworzenie z niego zwykłego skrótu kryptograficznego i użycie go jako klucza szyfrowania. W projektowaniu metod szyfrowania odpornych na ataki metodą brutalnej siły najważniejszą rolę odgrywają funkcje derywujące klucze. Użycie tylko kryptograficznego algorytmu mieszającego typu MD5 albo SHA1 sprawia, że klucz można złamać metodą brutalnej siły lub słownikową przy użyciu niewielkiego klastra komputerów, a nawet w niektórych przypadkach potężnego jednego komputera stacjonarnego. Niektóre rządy dysponują nawet możliwościami zaprojektowania i wyprodukowania specjalnych układów elektronicznych do wykonywania ataków metodą siłową, które mogą znacząco przyspieszyć operację łamania hasła. Proste mieszanie hasła jest słabym rozwiązaniem nawet dla zwykłych programów, nie mówiąc już o aplikacjach, które mogłyby zainteresować jakiś rząd.

Funkcje derywacyjne kluczy w porównaniu z mieszaniem kryptograficznym i innymi technikami mają wiele zalet. Po pierwsze: funkcje te przepuszczają dane wejściowe przez serię iteracji kryptograficznych, w wyniku których powstaje klucz szyfrowania. W odróżnieniu od prostego skrótu kryptograficznego funkcja KDF może wykonać 1000, a nawet 10 000 powtórzeń. Każda iteracja obliczania klucza zwiększa ilość zużywanych cykli procesora, co znacznie utrudnia przeprowadzenie skutecznego ataku metodą brutalnej siły. Weźmy na przykład klucz derywowany, którego wygenerowanie na urządzeniu zajmuje jedną sekundę. Podczas logowania do aplikacji przy użyciu prawidłowego hasła to sekundowe opóźnienie byłoby praktycznie niezauważalne, ale atak metodą brutalnej siły zostałby przez to znacznie bardziej wydłużony, niż gdyby użyto tylko prostego skrótu. Obliczenia dla każdej zgadywanej wartości zajmowałyby około jednej sekundy.

Kolejną zaletą funkcji derywacyjnej jest to, że może ona przedłużyć lub skrócić hasło, aby wygenerować klucz o żądanej długości. Dzięki temu nawet z czterocyfrowego hasła można by było utworzyć klucz 128- lub 256-bitowy albo o jeszcze jakimś innym rozmiarze.

PBKDF2 (ang. *password-based key derivation function*) to funkcja derywująca klucze opisana w specyfikacji PKCS algorytmu RSA jako funkcja derywująca klucze szyfrowania z haseł. Używa się jej w wielu popularnych implementacjach szyfrowania, wliczając program File Vault firmy Apple, TrueCrypt oraz WPA/WPA2 do zabezpieczania sieci WiFi. Funkcja PBKDF2 przyjmuje na wejściu hasło i tworzy przy jego użyciu klucz szyfrowania, stosując żądaną liczbę iteracji.

W kryptografii słowem **sól** określa się szereg bitów mających za zadanie utrudnić przeprowadzenie niektórych rodzajów ataków kryptoanalitycznych, takich jak ataki słownikowe wykonywane przy użyciu tęczowych tablic. Gdy do hasła zostanie dodana domieszka soli, wtedy to samo hasło w innym miejscu da inny klucz. Sposób uzyskiwania soli leży całkowicie w gestii programisty. Do wersji systemu iOS 5 jako soli używano niepowtarzalnego numeru sprzętowego UDID urządzenia. Dzięki temu zaszyfrowany klucz pasował tylko wtedy, gdy algorytm był uruchamiany na tym samym urządzeniu, na którym dokonano szyfrowania (pod warunkiem, że atakujący go nie sfałszował). Gdy w systemie iOS 5 zabroniono używania tego identyfikatora do tych celów, programiści byli zmuszeni poszukać czegoś innego. Padło na adres MAC karty sieciowej urządzenia, który jest niepowtarzalny i obecny w każdym urządzeniu. Na listingu 10.6 jest przedstawiona przykładowa metoda, która pobiera tę informację i zwraca ją jako obiekt klasy `NSString` gotowy do użytku jako sól.

Listing 10.6. Pobieranie adresu MAC karty sieciowej urządzenia

```
#import <Foundation/Foundation.h>
#include <openssl/evp.h>

#include <sys/socket.h>
#include <sys/sysctl.h>
#include <net/if.h>
#include <net/if_dl.h>

- (NSString *)query_mac {
    int mib[6];
    size_t len;
    char *buf;
    unsigned char *ptr;
    struct if_msghdr *ifm;
    struct sockaddr_dl *sdl;
```

```
    mib[0] = CTL_NET;
    mib[1] = AF_ROUTE;
    mib[2] = 0;
    mib[3] = AF_LINK;
    mib[4] = NET_RT_IFLIST;

    if ((mib[5] = if_nametoindex("en0")) == 0)
        return NULL;

    if (sysctl(mib, 6, NULL, &len, NULL, 0) < 0)
        return NULL;

    if ((buf = malloc(len)) == NULL)
        return NULL;

    if (sysctl(mib, 6, buf, &len, NULL, 0) < 0)
        return NULL;

    ifm = (struct if_msghdr *)buf;
    sdl = (struct sockaddr_dl *)(ifm + 1);
    ptr = (unsigned char *)LLADDR(sdl);

    NSString *out = [ NSString
        stringWithFormat:@ "%02X:%02X:%02X:%02X:%02X:%02X",
        *ptr, *(ptr+1), *(ptr+2), *(ptr+3), *(ptr+4), *(ptr+5) ];
    free(buf);

    return out;
}
```

Gdy wartość soli jest ściśle związana z urządzeniem, tak jak w tym przypadku, zaszyfrowanych danych nie da się odczytać z kopii na innym urządzeniu. W zależności od potrzeby może to być dokładnie to, czego szukasz, lub wręcz odwrotnie. Jeżeli dane muszą dać się odczytać na dowolnym urządzeniu, sól można generować losowo za pierwszym razem przy ustawianiu hasła i przechowywać wraz z zaszyfrowanym kluczem głównym na urządzeniu.

Niezależnie od sposobu wygenerowania soli jej wartości, hasła i liczby iteracji można użyć do wygenerowania klucza szyfrowania, przy użyciu którego następnie można zaszyfrować klucz główny. `PKCS5_PBKDF2_HMAC_SHA1` jest popularną funkcją PBKDF2 dostępną w OpenSSL. Na listingu 10.7 przedstawiona jest implementacja dla systemu iOS napisana przy użyciu biblioteki Common Crypto.

Listing 10.7. Implementacja funkcji PKCS5_PBKDF2_HMAC_SHA1

```
#include <CommonCrypto/CommonDigest.h>
#include <CommonCrypto/CommonHMAC.h>
#include <CommonCrypto/CommonCryptor.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int PKCS5_PBKDF2_HMAC_SHA1(
    const char *pass,
    int passlen,
    const unsigned char *salt,
    int saltlen,
    int iter,
    int keylen,
    unsigned char *out)
{
    unsigned char digtmp[CC_SHA1_DIGEST_LENGTH], *p, itmp[4];
    int cplen, j, k, tkeylen;
```

```
    unsigned long i = 1;
    CCHmacContext hctx;
    p = out;
    tkeylen = keylen;

    if (!pass)
        passlen = 0;
    else if (passlen == -1)
       passlen = strlen(pass);

    while(tkeylen) {
        if (tkeylen > CC_SHA1_DIGEST_LENGTH)
            cplen = CC_SHA1_DIGEST_LENGTH;
        else
            cplen = tkeylen;

        itmp[0] = (unsigned char)((i >> 24) & 0xff);
        itmp[1] = (unsigned char)((i >> 16) & 0xff);
        itmp[2] = (unsigned char)((i >> 8) & 0xff);
        itmp[3] = (unsigned char)(i & 0xff);
        CCHmacInit(&hctx, kCCHmacAlgSHA1, pass, passlen);
        CCHmacUpdate(&hctx, salt, saltlen);
        CCHmacUpdate(&hctx, itmp, 4);
        CCHmacFinal(&hctx, digtmp);
        memcpy(p, digtmp, cplen);
        for (j = 1; j < iter; j++) {
            CCHmac(kCCHmacAlgSHA1, pass, passlen, digtmp,
                CC_SHA1_DIGEST_LENGTH, digtmp);
            for(k = 0; k < cplen; k++)
                p[k] ^= digtmp[k];
        }
        tkeylen-= cplen;
        i++;
        p+= cplen;
    }
    return 1;
}
```

Przy użyciu tej implementacji można derywować klucz z hasła i soli. Jako argumenty wywołania tej funkcji należy przekazać hasło, długość hasła, sól, długość soli, liczbę iteracji, rozmiar klucza oraz wskaźnik na alokowany bufor:

```
NSString *device_id = [ myObject query_mac ];
unsigned char out[16];
char *passphrase = "secret!";

int r = PKCS5_PBKDF2_HMAC_SHA1(
        passphrase,
        strlen(passphrase),
        [ device_id UTF8String ],
        strlen([ device_id UTF8String]),
        10000, 16, out);
```

W powyższym przykładzie została ustawiona liczba 10 000 iteracji, co oznacza, że zanim funkcja `PKCS5_PBKDF2_HMAC_SHA1` zwróci klucz, najpierw wykona na nim 10 000 operacji. Wartość tę można zwiększyć albo zmniejszyć w zależności od tego, ile zasobów procesora chce się zużyć do wygenerowania klucza (co ma bezpośrednie przełożenie na to, ile zasobów procesora będzie potrzebnych do wykonania ataku brutalną siłą). W typowym iPadzie pierwszej generacji wykonanie 10 000 powtórzeń zajmuje około jednej sekundy. Należy wziąć to pod uwagę, planując ergonomię korzystania z urządzenia.

Wygenerowanego klucza można następnie użyć do zaszyfrowania klucza głównego. Następnie zaszyfrowany klucz główny i sól można zapisać na urządzeniu. Na listingu 10.8 przedstawiony jest program szyfrujący klucz główny przy użyciu klucza derywowanego przy użyciu funkcji `PKCS5_PBKDF2_HMAC_SHA1`.

Listing 10.8. Funkcja szyfrująca klucz główny przy użyciu funkcji PBKDF2

```
int encrypt_master_key(
    unsigned char *dest,
    const unsigned char *master_key,
    size_t key_len,
    const char *passphrase,
    const unsigned char *salt,
    int slen
) {
    CCCryptorStatus status;
    unsigned char cipherKey[key_len];
    unsigned char cipherText[key_len + kCCBlockSizeAES128];
    size_t nEncrypted;
    int r;


    r = PKCS5_PBKDF2_HMAC_SHA1(
        passphrase, strlen(passphrase),
        salt, slen,
        10000, key_len, cipherKey);

    if (r < 0)
        return r;

    status = CCCrypt(kCCEncrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        cipherKey,
        key_len,
        NULL,
        master_key, key_len,
        cipherText, sizeof(cipherText),
        &nEncrypted);
    if (status != kCCSuccess) {
        printf("Funkcja CCCrypt() zwróciła błąd %d\n", status);
        return status;
    }

    memcpy(dest, cipherText, key_len);
    return 0;
}
```

Geoszyfrowanie

Mimo że główny klucz szyfrowania jest już zabezpieczony hasłem, jego ochronę można jeszcze wzmocnić dodatkowymi technikami. Biorąc pod uwagę, że większość pracowników firm używa haseł łatwych do złamania, warto zwiększyć poziom bezpieczeństwa poprzez zastosowanie technik lokalizacyjnych. **Geoszyfrowanie** (ang. *geo-encryption*) polega na wykorzystaniu do szyfrowania danych określonych lokalizacji geograficznych. Atakujący, chcąc rozszyfrować tak zaszyfrowane dane, musi znać współrzędne jakiegoś tajnego miejsca, np. jakiejś jednostki rządowej.

Szyfrowanie lokalizacyjne to zaawansowana technika, o wiele bardziej wyrafinowana niż zwykła logika kodu dotycząca lokalizacji. Wiesz już z pierwszej części tej książki, że atakujący może z łatwością ominąć wewnętrzne testy logiczne, a nawet sforsować je metodą brutalnej siły. Współrzędne GPS można sfałszować tak, aby urządzenie „myślało", że znajduje się w określonym miejscu. Ponadto testy sprawdzające lokalizację wymagają zapisania w urządzeniu współrzędnych GPS wybranego tajnego miejsca, które same w sobie mogą być łakomym kąskiem dla przestępcy. Gdy hakerowi uda się złamać zabezpieczenia urządzenia, to jego następnym krokiem może być fizyczny atak na miejsce, którego lokalizację uda mu się odnaleźć. Wszystkie te techniki dają tylko pozorne poczucie bezpieczeństwa.

Trudnością w opracowywaniu dobrego systemu kryptograficznego z wykorzystaniem geoszyfrowania jest entropia. Wyobraźmy sobie, że haker dokonuje ataku słownikowego z wykorzystaniem zamiast słów wszystkich możliwych par wartości długości i szerokości geograficznych. Kluczem w geoszyfrowaniu jest fizyczna lokalizacja, z którą związany jest szyfr, w związku z czym w tego rodzaju szyfrowaniu nie należy wykorzystywać lokalizacji zapisanych w książce adresowej urządzenia ani pamięci map Google. Atakujący na podstawie informacji o tym, gdzie zdobył urządzenie, a także danych pochodzących z samego urządzenia może odgadnąć przybliżoną lokalizację użytą do szyfrowania w promieniu około 32 kilometrów. Jeżeli zaszyfrowany plik został powiązany z lokalizacją z dokładnością do jednego metra, to w tym okręgu o promieniu 32 kilometrów istnieje około miliarda możliwych kombinacji lokalizacyjnych. Liczba ta znacznie się zmniejsza wraz ze stopniem dokładności lokalizacji. Przy 10-metrowym promieniu liczba kombinacji szerokości i długości geograficznych spada do stu milionów, a przy 100 metrach jest ich już tylko 10 milionów. Wykonanie ataku metodą brutalnej siły przy 10 milionach kombinacji zajęłoby bardzo mało czasu — zapewne zaledwie kilka godzin. Dlatego w technice tej ważną rolę odgrywają funkcje derywujące klucze.

Używając funkcji derywacji kluczy w połączeniu z geoszyfrowaniem, można sprawić, że ilość obliczeń potrzebna do wygenerowania klucza w znacznym stopniu udaremni ataki, nie powodując zbyt dużych niedogodności dla użytkownika. Weźmy na przykład funkcję PBKDF2, o której była mowa wcześniej w tym rozdziale. Jeśli zastosuje się dużą liczbę iteracji, aby wygenerowanie klucza zajmowało od 5 do 10 sekund, atak metodą słownikową zajmowałby bardzo dużo czasu. Przy 10 milionach kombinacji, jeśli każda próba zajmuje pięć sekund, na rozszyfrowanie urządzenia potrzeba 578 dni. Gdyby zwiększyć poziom zabezpieczeń i ograniczyć promień do 10 metrów, to atak w obszarze o promieniu 32 kilometrów zająłby około 15 lat.

Aby skrócić ilość czasu potrzebnego na wykonanie ataku, haker musiałby zdezasemblować i przenieść algorytm derywacji klucza oraz kod obiektu deszyfrującego dane do potężniejszego systemu. To także wymaga czasu. Przyjmując, że nowoczesne urządzenia z systemem iOS są wyposażone w szybkie dwurdzeniowe procesory, można spodziewać się tylko niewielkiego przyspieszenia, chyba że do ataku zostanie użyty cały klaster komputerów. Im krótszy okres przydatności chronionych danych, tym więcej zasobów trzeba mieć, aby przeprowadzić skuteczny atak.

Entropię można dodatkowo powiększyć poprzez dodanie czynnika **czasu**. Rozszerzając szyfrowanie geograficzne o dodatkowy parametr, jakim jest czas, można jeszcze bardziej zmniejszyć okno, w którym możliwe jest rozszyfrowanie danych. Jeśli na przykład dane można będzie rozszyfrować tylko w czasie jednej godziny w ciągu doby, czas potrzebny na przeprowadzenie ataku wydłuży się 24-krotnie. Ta technika szyfrowania może być użyta do zabezpieczenia materiałów, których tajność zależy od czasu i lokalizacji, np. hitu filmowego. Skrócenie czasu do pół godziny powoduje wydłużenie ataku 48 razy.

Aby zastosować geoszyfrowanie w swojej aplikacji, możesz użyć znanej Ci już funkcji PBKDF2. Określ najlepszą liczbę iteracji, biorąc pod uwagę wymagany poziom zabezpieczeń. Na przykład w iPhonie 4 wykonanie 650 000 iteracji zajmuje około pięciu sekund. Pamiętaj jednak, że przestępca może mieć nowszy i znacznie szybszy model urządzenia. Na listingu 10.9 przedstawiona jest zmodyfikowana wersja funkcji wywołująca funkcję PBKDF2 w celu zaszyfrowania klucza głównego przy użyciu danych GPS jako hasła i wykonująca 650 000 iteracji w celu wygenerowania klucza.

Listing 10.9. Funkcja geoszyfrowania szyfrująca klucz główny przy użyciu współrzędnych GPS

```
int geo_encrypt_master_key(
    unsigned char *dest,
    const unsigned char *master_key,
    size_t key_len,
    const char *geo_coordinates,
    const unsigned char *salt,
    int slen
) {
    CCCryptorStatus status;
    unsigned char cipherKey[key_len];
    unsigned char cipherText[key_len + kCCBlockSizeAES128];
    size_t nEncrypted;
    int r;

    r = PKCS5_PBKDF2_HMAC_SHA1(
        geo_coordinates, strlen(geo_coordinates),
        salt, slen,
        650000, key_len, cipherKey);

    if (r < 0)
        return r;

    status = CCCrypt(kCCEncrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        cipherKey,
        key_len,
        NULL,
        master_key, key_len,
        cipherText, sizeof(cipherText),
        &nEncrypted);
    if (status != kCCSuccess) {
        printf("Funkcja CCCrypt() zwróciła błąd %d\n", status);
        return status;
    }

    memcpy(dest, cipherText, key_len);
    return 0;
}
```

Funkcji tej należy podać parę współrzędnych zamiast hasła.

```
unsigned char encrypted_master_key[16];
char *coords = "30.2912,-97.7385";

geo_encrypt_master_key(
    encrypted_master_key,
    master_key, kCCKeySizeAES128,
    coords, salt, salt_len);
```

Współrzędne GPS można zaokrąglić do najbliższego miejsca dziesiętnego w zależności od promienia obszaru blokady (tabela 10.1). Pamiętaj, że moduł GPS w większości urządzeń z systemem iOS wskazuje współrzędne z ograniczoną precyzją, zazwyczaj do około 10 metrów.

Tabela 10.1. Tabela zaokrągleń wartości GPS

Jednostki	Precyzja szerokości geograficznej	Precyzja długości geograficznej
1,0	111 km	67,5 km
0,1	11 km	6,75 km
0,01	1,11 km	676 m
0,001	111 m	67,6 m
0,0001	11,1 m	6,76 m
0,00001	1,1 m	67,6 cm

Geoszyfrowanie z hasłem

Bezpieczeństwo geoszyfrowania jest całkowicie zależne od utrzymania w tajemnicy współrzędnych geograficznych wybranego miejsca (i ewentualnie czasu). Jeśli zostanie użyta funkcja derywacyjna kluczy zajmująca odpowiednią ilość czasu, atakujący będzie potrzebował bardzo precyzyjnych informacji na temat lokalizacji geograficznej wykorzystanej do szyfrowania. Atak można jeszcze bardziej utrudnić, dodając do tego wszystkiego hasło. Należy zmodyfikować funkcję PBKDF2 w taki sposób, aby generowała dwa klucze szyfrowania: jeden na podstawie hasła i drugi na podstawie współrzędnych geograficznych. Następnie klucze te łączy się za pomocą operacji XOR w jeden klucz, którego z kolei używa się do zaszyfrowania klucza głównego (listing 10.10).

Listing 10.10. Funkcja szyfrująca klucz główny przy użyciu hasła i współrzędnych GPS

```
int geo_encrypt_master_key(
    unsigned char *dest,
    const unsigned char *master_key,
    size_t key_len,
    const char *geocoordinates,
    const char *passphrase,
    const unsigned char *salt,
    int slen
) {
    CCCryptorStatus status;
    unsigned char cKey1[key_len], cKey2[key_len];
    unsigned char cipherText[key_len + kCCBlockSizeAES128];
    size_t nEncrypted;
    int r, i;
    /* Derywacja klucza z hasła */
    r = PKCS5_PBKDF2_HMAC_SHA1(
        passphrase, strlen(passphrase),
        salt, slen,
        10000, key_len, cKey1);
    if (r < 0)
        return r;
    /* Derywacja klucza z danych GPS */
    r = PKCS5_PBKDF2_HMAC_SHA1(
        geocoordinates, strlen(geocoordinates),
        salt, slen,
        650000, key_len, cKey2);
    if (r < 0)
        return r;
```

```
    /* Połączenie kluczy operacją XOR */
    for(i=0;i<key_len;++i)
        cKey1[i] ^= cKey2[i];

    status = CCCrypt(kCCEncrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        cKey1,
        key_len,
        NULL,
        master_key, key_len,
        cipherText, sizeof(cipherText),
        &nEncrypted);
    if (status != kCCSuccess) {
        printf("Funkcja CCCrypt() zwróciła błąd %d\n", status);
        return status;
    }
    memcpy(dest, cipherText, key_len);
    return 0;
}
```

Funkcję tę wywołuje się w podobny sposób jak pozostałe funkcje `encrypt_master_key` przedstawione w tym rozdziale. W tym przypadku należy tylko podać zarówno współrzędne GPS, jak i hasło.

```
unsigned char encrypted_master_key[16];
char *coords = "30.2912,-97.7385";
char *passphrase = "passphrase";

geo_encrypt_master_key(
    encrypted_master_key,
    master_key, kCCKeySizeAES128,
    coords,
    passphrase,
    salt, salt_len);
```

Aby dodać czynnik czasu, należy tylko wkomponować w kod godzinę, pół godziny, kwadrans lub dowolny inny okres.

```
unsigned char encrypted_master_key[16];
char *coords = "30.2912,-97.7385,05:00";
char *passphrase = "passphrase";

geo_encrypt_master_key(
    encrypted_master_key,
    master_key, kCCKeySizeAES128,
    coords,
    passphrase,
    salt, salt_len);
```

Teraz atakujący musi znać (lub zdobyć) hasło, współrzędne GPS i czas, aby rozszyfrować klucz główny.

Dzielenie kluczy na serwerze

Podobnie jak za pomocą danych geograficznych algorytm szyfrowania można wzmocnić także poprzez zastosowanie kluczy przechowywanych na serwerze, aby przed rozszyfrowaniem danych urządzenie musiało najpierw uwierzytelnić się w zdalnym systemie. W tym przypadku generuje się dwa klucze, które miesza się za pomocą operacji XOR i za pomocą uzyskanego klucza szyfruje się klucz główny. Jeden z tych kluczy jest generowany przy użyciu podanego

przez użytkownika hasła. Drugi natomiast jest generowany losowo i zapisywany na zaufanym zdalnym serwerze przy pierwszym instalowaniu aplikacji. Po uruchomieniu programu użytkownik wpisuje hasło, aby wygenerować swoją połowę klucza, ale dodatkowo musi uwierzytelnić się na serwerze, aby uzyskać drugą połowę. Dzięki temu ani serwer, ani urządzenie nie ma wszystkiego, co jest potrzebne do rozszyfrowania danych aplikacji. To dodatkowo utrudnia atakowanie haseł, ponieważ samo hasło nie wystarcza do rozszyfrowania danych. Przestępca musi nie tylko złamać hasło, ale dodatkowo zabezpieczenia serwera zawierającego drugą połowę klucza. Kolejną korzyścią jest możliwość usunięcia klucza z serwera, gdy zostanie odkryte, że urządzenie skradziono lub że złamano jego zabezpieczenia. W rozdziale 12. poznasz techniki reagowania na próby szperania w zabezpieczeniach i sposoby ich praktycznego zastosowania do ochrony danych, które nie są w danej chwili w użyciu. Usunięcie klucza z serwera jest efektywnym sposobem na uniemożliwienie eskalacji problemu.

Opisywana technika ma jednak też wady. Kradzież danych z urządzenia może nastąpić dopiero po ich rozszyfrowaniu. Podczas używania danych w aplikacji w pamięci muszą być przechowywane te dane albo klucze szyfrowania. Jednak technika ta w niektórych zastosowaniach sprawdza się doskonale.

Na listingu 10.11 przedstawiony jest program generujący dwa klucze. Działa on podobnie do wcześniej prezentowanych programów. Pierwszy klucz, `userKey`, jest generowany z podanego przez użytkownika hasła. Drugi, `serverKey`, jest generowany losowo.

Listing 10.11. Funkcja generująca parę kluczy

```
#include <CommonCrypto/CommonCryptor.h>
#include <string.h>
#include <stdio.h>

int split_encrypt_master_key(
    unsigned char *encryptedMasterKey, /* Zapisywany w buforze */
    unsigned char *serverKey, /* Zapisywany w buforze */
    const unsigned char *master_key,
    size_t key_len,
    const char *passphrase,
    const unsigned char *salt,
    int slen
) {
    CCCryptorStatus status;
    unsigned char userKey[key_len];
    unsigned char cipherText[key_len + kCCBlockSizeAES128];
    size_t nEncrypted;
    int r, i;

    /* Derywacja klucza użytkownika z hasła */
    r = PKCS5_PBKDF2_HMAC_SHA1(
        passphrase, strlen(passphrase),
        salt, slen,
        10000, key_len, userKey);
    if (r < 0)
        return r;

    /* Generowanie losowego klucza, zapis w serverKey */
    for(i=0;i<key_len;++i)
        serverKey[i] = arc4random() % 255;

    /* Połączenie XOR kluczy w kluczu userKey */
    for(i=0;i<key_len;++i)
        userKey[i] ^= serverKey[i];
```

```
    status = CCCrypt(kCCEncrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        userKey,
        key_len,
        NULL,
        master_key, key_len,
        cipherText, sizeof(cipherText),
        &nEncrypted);
    if (status != kCCSuccess) {
        printf("CCCrypt() failed with error %d\n", status);
        return status;
    }

    memcpy(encryptedMasterKey, cipherText, key_len);
    return 0;
}
```

Aby użyć tej funkcji, należy alokować dwa bufory: jeden na zaszyfrowany klucz główny, a drugi na klucz serwerowy. Jako argumenty wywołania podaje się hasło, sól i inne informacje.

```
unsigned char encryptedMasterKey[kCCKeySizeAES128];
unsigned char serverKey[kCCKeySizeAES128];

split_encrypt_master_key(
    encryptedMasterKey,
    serverKey,
    master_key,
    kCCKeySizeAES128,
    passphrase,
    salt,
    slen);
```

Po zakończeniu działania przez funkcję klucz serwerowy będzie zapisany w alokowanym dla niego buforze. Powinien on zostać zarejestrowany na serwerze i usunięty z urządzenia przy pierwszym uruchomieniu aplikacji. Między urządzeniem a serwerem musi znajdować się mechanizm uwierzytelniający zabezpieczający operację przesyłania klucza z powrotem z serwera na urządzenie w przyszłości.

Ponieważ klucze są wymieniane podczas pierwszego uruchomienia aplikacji, należy się upewnić, że urządzenie nie jest w tym momencie w żaden sposób zaatakowane. Klucz z serwera można dostarczać także na inne sposoby. Wstawienie do aplikacji dodatkowego pola tekstowego na klucz serwerowy może pomóc zapobiec atakom podmiany kluczy.

Zabezpieczanie pamięci

Jak już wiesz, przechowywanie kluczy szyfrowania i innych danych w pamięci jest niebezpieczne, ponieważ jeśli urządzenie jest zaatakowane, to mogą one zostać skradzione podczas wczytywania ich do pamięci. W nowszych wersjach systemu iOS wbudowane są mechanizmy losowej zmiany rozkładu przestrzeni adresowej, które mają za zadanie ukrywać lokalizacje fragmentów pamięci na tyle skutecznie, że zanim haker je znajdzie, to spowoduje awarię programu. Niestety, jak widzieliśmy, zmienne egzemplarzowe języka Objective-C można z łatwością znaleźć poprzez mapowanie systemu wykonawczego, co w znacznym stopniu ogranicza przydatność techniki ASLR. Poniżej znajdują się wskazówki, jak zabezpieczyć pamięć:

- Nie zapisuj niczego w pamięci, dopóki użytkownik nie uwierzytelni się i dane nie zostaną rozszyfrowane. Dopóki użytkownik nie wpisze hasła, nie powinno się nawet zapisywać haseł, danych uwierzytelniających ani żadnych innych informacji. Jeśli jest to możliwe, to znaczy, że szyfrowanie w aplikacji jest źle zaimplementowane.
- Nie przechowuj kluczy szyfrowania ani innych ważnych danych w zmiennych egzemplarzowych w języku Objective-C, ponieważ można łatwo uzyskać do nich dostęp. Zamiast tego ręcznie alokuj dla nich pamięć. To nie uniemożliwi hakerowi podpięcia się pod Twoją aplikację za pomocą debugera, ale utrudni przeprowadzenie ataku. Ataki dokonywane podczas używania urządzenia są najczęściej przeprowadzane przez automaty, a nie ludzi. Oprogramowanie takie najpierw sięga po najprostsze środki działania i jeśli nie zostało specjalnie zaprojektowane pod kątem konkretnej aplikacji, rzadko kiedy znajduje inne dane niż te zapisane w zmiennych egzemplarzowych.
- Nie zapisuj w zmiennych egzemplarzowych wskaźników na klucze szyfrowania i inne ważne dane.
- Jeśli to tylko możliwe, usuwaj dane z pamięci natychmiast, gdy przestają być potrzebne. Jeżeli na przykład program zostaje przeniesiony do działania w tle w trybie zawieszonym albo użytkownik zamknie jakiś plik, klucze szyfrowania potrzebne do korzystania z tych zasobów powinny zostać usunięte.

Czyszczenie pamięci

Jeśli używane dane są zapisywane w pamięci, to po zakończeniu pracy powinny być zawsze usuwane. Takie informacje jak klucze szyfrowania, numery kart kredytowych itp. nie muszą zalegać na dysku podczas działania aplikacji, a pozostawienie ich stanowi tylko dodatkowe ryzyko utraty. Na szczęście większość klas z biblioteki Foundation umożliwia odwoływanie się do wskaźników na rzeczywiste dane, co pozwala na ich usunięcie przed zwolnieniem.

W obiektach klasy `NSData` dostępna jest metoda `bytes` pozwalająca tworzyć wskaźniki na dane w pamięci. Przy użyciu funkcji `memset` dane te można z łatwością nadpisać:

```
memset([ myData bytes ], 0, [ myData length ]);
```

Obiekty klasy `NSString` są nieco bardziej skomplikowane. Wszystkie metody dostępowe tych obiektów zwracają kopię danych, a nie same dane. Jednak klasa `NSString` jest zamienna z klasą `CFString` będącą składnikiem biblioteki Core Foundation Apple. Funkcja C `CFStringGetCStringPtr` zwraca wskaźnik na dane zamiast kopii danych, które są zwracane przez metody dostępowe klasy `NSString`. Dane wskazywane przez wskaźnik można usunąć przed zwolnieniem łańcucha.

Poniżej znajduje się przykładowy kod usuwający dane w obiekcie `NSString` i drukujący usuwaną treść przy użyciu metody `UTF8String`, aby pokazać, że łańcuch rzeczywiście został usunięty.

```
unsigned char *text;
char x[5];

strcpy(x, "1234");
NSString *myString = [ [ NSString alloc ] initWithFormat: @"%s", x ];
text = (unsigned char *) CFStringGetCStringPtr((CFStringRef) myString,
            CFStringGetSystemEncoding());
printf("Oryginalny tekst: %s\n", text);
```

```
memset(text, 0, [ myString length ]);
printf("Nowy tekst:%s\n", [ myString UTF8String ]);
[ myString release ];
```

Jeśli używasz kodowania UTF-16, to zamiast `CFStringGetCStringPtr` użyj metody `CFStringGetCharactersPtrfunction`.

Kryptografia klucza publicznego

Wielu programistów nieświadomych wad protokołu SSL używa go jako jedynego zabezpieczenia w swoich aplikacjach. SSL to oczywiście bardzo ważny składnik zabezpieczeń technologii handlu elektronicznego, ale nie może być jedynym używanym zabezpieczeniem. Złośliwe oprogramowanie i ataki typu człowiek pośrodku zazwyczaj nie zaglądają do pamięci wewnątrz aplikacji i mogą tylko podsłuchiwać sesje SSL. W takich przypadkach dodatkowa warstwa szyfrowania może zapewnić bezpieczeństwo danym. Ponadto wiele rządów, w tym rządy USA, Chin, a także firmy komunikacyjne należące do różnych krajów mają własne instytucje certyfikacyjne, których certyfikaty są wbudowane w komponenty sieciowe systemu iOS. Jeżeli projektujesz aplikację, która może być podsłuchiwana przez obcy rząd, certyfikaty te mogą zostać wykorzystane do podszycia się pod legalne witryny internetowe w celu przechwycenia ruchu. Dodaj do tego techniki podsłuchu, jakimi dysponują rządy, sprzęt do maskowania podsłuchów i wiele innych technik szpiegowskich, a od razu dostrzeżesz, dlaczego nie należy polegać wyłącznie na protokole SSL. Dane należy chronić przy użyciu tego protokołu i dodatkowych warstw szyfrowania, takich jak kryptografia klucza publicznego.

Kryptografia klucza publicznego to rodzaj kryptografii asymetrycznej, w której do szyfrowania i deszyfrowania danych używane są osobne klucze. W technice tej urządzenie wysyłające dane zna klucz publiczny odbiorcy. Klucz ten jest czymś w rodzaju wzoru na szyfrowanie przesyłanych informacji. W większości przypadków odbiorcą jest serwer przechowujący zdalne zasoby, do których dostępu potrzebuje nasza aplikacja — np. system finansowy. Serwer zna odpowiedni klucz prywatny, który służy do rozszyfrowania danych. Najważniejsza jest bezpieczna wymiana kluczy, która powinna odbyć się przy pierwszym użyciu aplikacji.

Na listingu 10.12 przedstawiony jest przykład użycia frameworku Security firmy Apple do zaszyfrowania i rozszyfrowania wiadomości przy użyciu pary klucz publiczny – klucz prywatny. W prawdziwym programie szyfrowanie odbywa się na urządzeniu, natomiast deszyfrowanie na serwerze. Tutaj obie operacje są wykonywane w tym samym miejscu.

Listing 10.12. Szyfrowanie i deszyfrowanie przy użyciu klucza publicznego (seccrypt.m)

```
#import <Foundation/Foundation.h>
#import <Security/Security.h>

void example_pki( ) {
    SecKeyRef publicKey;
    SecKeyRef privateKey;

    CFDictionaryRef keyDefinitions;
    CFTypeRef keys[2];
    CFTypeRef values[2];

    /* Parametry nowej pary kluczy */
    keys[0] = kSecAttrKeyType;
    values[0] = kSecAttrKeyTypeRSA;
```

```
    keys[1] = kSecAttrKeySizeInBits;
    int iByteSize = 1024;
    values[1] = CFNumberCreate(NULL, kCFNumberIntType, &iByteSize);

    keyDefinitions = CFDictionaryCreate(
        NULL, keys, values, sizeof(keys) / sizeof(keys[0]), NULL, NULL );

    /* Wygenerowanie nowej pary kluczy */
    OSStatus status = SecKeyGeneratePair(keyDefinitions,
        &publicKey, &privateKey);

    /* Przykładowe dane uwierzytelniające wysyłane do serwera */
    unsigned char *clearText = "username=USERNAME&password=PASSWORD";
    unsigned char cipherText[1024];
    size_t buflen = 1024;

    /* Szyfrowanie: wykonywane na urządzeniu */
    status = SecKeyEncrypt(
        publicKey, kSecPaddingNone, clearText, strlen(clearText) + 1,
        &cipherText[0], &buflen);

    /* Deszyfrowanie: wykonywane na serwerze */
    unsigned char decryptedText[buflen];
    status = SecKeyDecrypt(privateKey, kSecPaddingNone, &cipherText[0],
        buflen, &decryptedText[0], &buflen);
}
```

Skompiluj ten program za pomocą kompilatora krzyżowego z Xcode i dołącz bibliotekę `Security`.

```
$ export PLATFORM=/Developer/Platforms/iPhoneOS.platform
$ $PLATFORM/Developer/usr/bin/arm-apple-darwin10-llvm-gcc-4.2 \
    -c -o seccrypt seccrypt.m \
    -isysroot $PLATFORM/Developer/SDKs/iPhoneOS5.0.sdk \
    -framework Foundation -framework Security -lobjc
```

W kodzie tym generowana jest para kluczy publicznego i prywatnego, które są używane na tym samym urządzeniu. W realnym świecie klucz prywatny znany jest tylko urządzeniu deszyfrującemu. Wygenerowane klucze są zazwyczaj zapisywane na dysku, a następnie wysyłane do nadawcy i odbiorcy. W Twojej aplikacji możesz nawet klucz publiczny zaszyfrować przy użyciu klucza głównego wygenerowanego z hasła użytkownika, aby uniemożliwić jego zdobycie przez niepowołaną osobę bez znajomości klucza użytkownika. Nadawca wiadomości szyfruje dane wysyłane do odbiorcy za pomocą klucza publicznego. Rozszyfrować je może tylko odbiorca mający klucz prywatny. Dzięki temu klucz publiczny może być udostępniony w aplikacjach oferowanych w App Store bez ryzyka naruszenia zabezpieczeń.

Poniższych funkcji można użyć do importowania i eksportowania elementów `SecKeyRef` reprezentujących klucze przy użyciu klasy `NSData`:

```
NSData *exportKey(SecKeyRef key) {
    SecItemImportExportKeyParameters params;
    CFMutableArrayRef keyUsage
        = (CFMutableArrayRef) [ NSMutableArray
            arrayWithObjects: kSecAttrCanEncrypt, kSecAttrCanDecrypt, nil ];
    CFMutableArrayRef keyAttributes
        = (CFMutableArrayRef) [ NSMutableArray array ];
    SecExternalFormat format = kSecFormatUnknown;
    CFDataRef keyData;
    OSStatus oserr;
    int flags = 0;

    memset(&params, 0, sizeof(params));
    params.version = SEC_KEY_IMPORT_EXPORT_PARAMS_VERSION;
```

```
    params.keyUsage = keyUsage;
    params.keyAttributes = keyAttributes;

    oserr = SecItemExport(key, format, flags, &params, &keyData);
    if (oserr) {
        fprintf(stderr, "Błąd SecItemExport\n", oserr);
        return nil;
    }
    return (NSData *) keyData;
}

SecKeyRef importKey(NSString *filename) {
    SecItemImportExportKeyParameters params;
    SecExternalItemType itemType = kSecItemTypeUnknown;
    SecExternalFormat format = kSecFormatUnknown;
    __block CFArrayRef items = NULL;
    SecKeyRef loadedKey;
    NSData *keyData;
    OSStatus oserr;
    int flags = 0;

    keyData = [ NSData dataWithContentsOfFile: filename ];

    memset(&params, 0, sizeof(params));
    params.keyUsage = NULL;
    params.keyAttributes = NULL;

    oserr = SecItemImport((CFDataRef) keyData, NULL, &format, &itemType,
        flags, &params, NULL, &items);
    if (oserr) {
        fprintf(stderr, "Błąd SecItemExport\n", oserr);
        exit(-1);
    }

    loadedKey = (SecKeyRef)CFArrayGetValueAtIndex(items, 0);
    return loadedKey;
}
```

Przy użyciu tych funkcji można napisać funkcję generującą losową parę kluczy i zapisującą ją na dysku.

```
void generateRandomKeyPair(NSString *filename) {
    SecKeyRef publicKey;
    SecKeyRef privateKey;

    CFDictionaryRef keyDefinitions;
    CFTypeRef keys[2];
    CFTypeRef values[2];

    /* Określenie parametrów nowej pary kluczy */
    keys[0] = kSecAttrKeyType;
    values[0] = kSecAttrKeyTypeRSA;

    keys[1] = kSecAttrKeySizeInBits;
    int iByteSize = 1024;
    values[1] = CFNumberCreate(NULL, kCFNumberIntType, &iByteSize);

    keyDefinitions = CFDictionaryCreate(
        NULL, keys, values, sizeof(keys) / sizeof(keys[0]), NULL, NULL );

    /* Generowanie nowej pary kluczy */
    OSStatus status = SecKeyGeneratePair(keyDefinitions,
        &publicKey, &privateKey);
```

```
    NSData *privateKeyData = exportKey(privateKey);
     [ privateKeyData writeToFile: filename atomically: NO ];

    NSData *publicKeyData = exportKey(publicKey);
     [ publicKeyData writeToFile:
         [ NSString stringWithFormat: @"%@.pub", filename ]
        atomically: NO ];
}
```

Teraz aplikacja może wczytać klucz publiczny z dysku, a następnie zaszyfrować wiadomość do odbiorcy. Poniżej znajduje się przykład ilustrujący sposób utworzenia losowej pary kluczy i wczytania klucza publicznego z dysku w celu zaszyfrowania wiadomości:

```
int main() {
    unsigned char clearText[1024];
    unsigned char cipherText[1024];
    size_t len = sizeof(cipherText);
    OSStatus status;
    int i;

    NSAutoreleasePool *pool = [ [ NSAutoreleasePool alloc ] init ];

    generateRandomKeyPair(@"mykeys");

    /* Szyfrowanie */
    SecKeyRef publicKey = importKey(@"mykeys.pub");
    strcpy(clearText, "username=USERNAME&password=PASSWORD");
    memset(cipherText, 0, sizeof(cipherText));
    CFShow(publicKey);

    status = SecKeyEncrypt(
        publicKey, kSecPaddingNone, clearText, strlen(clearText) + 1,
        &cipherText, &len);
    if (status != errSecSuccess) {
        NSLog(@"Szyfrowanie nie powiodło się: %d\n", status);
        return EXIT_FAILURE;
    }

    printf("Szyfr: ");
    for(i=0;i<strlen(clearText);++i)
        printf("%02x", cipherText[i]);
    printf("\n");
     [ pool release ];
}
```

Oczywiście klucz prywatny powinien być dostępny tylko po jednej stronie. Jeśli planujesz dwukierunkową komunikację przy użyciu tego rodzaju szyfrowania asymetrycznego, to pamiętaj, że dla każdego punktu końcowego powinna być wygenerowana osobna para kluczy.

Odbiorca komunikatu, którym najczęściej jest serwer, wczytuje posiadany klucz prywatny w celu rozszyfrowania przychodzących do niego wiadomości. Ilustruje to poniższy przykład:

```
/* Deszyfrowanie */
SecKeyRef privateKey = importKey(@"mykeys");
memset(clearText, 0, sizeof(clearText));
CFShow(privateKey);
status = SecKeyDecrypt(privateKey, kSecPaddingNone, &cipherText,
    len, &clearText, &len);
if (status != errSecSuccess) {
    NSLog(@"Deszyfrowanie nie powiodło się: %d\n", status);
    return EXIT_FAILURE;
}
printf("Czysty tekst: %s\n", clearText);
```

W aplikacjach desktopowych do szyfrowania i deszyfrowania danych należy używać parametru `kSecPaddingPKCS1` zamiast `kSecPaddingNone`. Natomiast w implementacjach dla systemu iOS należy używać opcji `kSecPaddingNone`.

Ćwiczenia

- Zaimplementuj funkcję PBKDF2 w swojej własnej aplikacji, aby zaszyfrować wszystkie główne klucze szyfrowania przy użyciu hasła.
- Mimo iż PBKDF2 to najczęściej używana funkcja derywacyjna kluczy w programach dla użytkowników domowych, istnieją jeszcze dwie inne popularne funkcje tego typu, które są bardziej odporne na ataki brutalną siłą. Pobierz i skompiluj biblioteki kryptograficzne `bcrypt` i `scrypt` i rozważ, na ile możliwe jest wykorzystanie ich w Twoich aplikacjach.
- Na podstawie zdobytej wiedzy na temat rozdzielonych kluczy napisz przykładową aplikację deszyfrującą wspólne dane między dwoma lub większą liczbą urządzeń z systemem iOS, ale tylko wówczas, gdy urządzenia te są w swoim zasięgu. Użyj biblioteki `GameKit` firmy Apple do wymiany kluczy przy użyciu kryptografii klucza publicznego.
- Napisz metodę usuwającą dane przyjmującą jako argument wskaźnik na obiekt klasy `NSString`. Metoda ta powinna pozyskiwać wskaźnik na bajty przechowywane w obiekcie i kasować ich treść.

Skorowidz

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ż

Programowanie

O'REILLY®

Łamanie i zabezpieczanie aplikacji w systemie iOS

iOS to obecnie jeden z najpopularniejszych systemów operacyjnych, wykorzystywany w urządzeniach firmy Apple. Jednak dzięki tej popularności jest on też łakomym kąskiem dla hakerów. Uzyskanie dostępu do danych przechowywanych w telefonie może mieć katastrofalne skutki. Dlatego jeżeli tworzysz aplikacje na platformę iOS, ta książka jest dla Ciebie pozycją obowiązkową.

Jak obronić się przed atakiem? Wszystkie niezbędne informacje znajdziesz w tym wyjątkowym podręczniku. W trakcie lektury dowiesz się, jak działają hakerzy, jak wyszukują słabe punkty aplikacji oraz jak modyfikują jej kod. Ponadto nauczysz się utrudniać śledzenie kodu Twojej aplikacji oraz bezpiecznie usuwać pliki (tak aby nie było możliwe ich odtworzenie). Wśród poruszanych tematów znajdziesz również te związane z transmisją danych: wykorzystanie protokołu SSL to nie wszystko, musisz zadbać także o to, żeby nie było możliwe przejęcie sesji SSL. Weź książkę do ręki i obroń się przed atakiem!

Dzięki tej książce:

- zrozumiesz, jak działają hakerzy
- zabezpieczysz swoją aplikację przed nieuprawnionymi zmianami
- ochronisz swoje bezpieczne połączenia
- bezpowrotnie usuniesz niepotrzebne pliki
- zagwarantujesz bezpieczeństwo danych użytkownikom Twojej aplikacji

Zadbaj o bezpieczeństwo danych użytkowników Twojej aplikacji!

ISBN 978-83-246-5147-4