

Twoja lektura obowiązkowa!



Łamanie i zabezpieczanie aplikacji w systemie iOS



O'REILLY®

Jonathan Zdziarski

Tytuł oryginału: Hacking and Securing iOS Applications

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-5147-4

© 2012 Helion S.A.

Authorized Polish translation of the English edition of Hacking and Securing iOS Applications ISBN 9781449318741 © 2012 Jonathan Zdziarski.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp.....	11
1. Wszystko, co wiesz, jest błędne	15
Mit monokultury	16
Model zabezpieczeń systemu iOS	18
Składniki modelu zabezpieczeń systemu iOS	19
Przechowywanie klucza i zamka w jednym miejscu	21
Hasła jako słabe zabezpieczenie	22
Ślady pozostawiane przez system niweczą szyfrowanie	24
Dane zewnętrzne również są zagrożone	24
Przechwytywanie ruchu	25
Kradzież danych jest... błyskawiczna	25
Nie ufaj nikomu, nawet własnym aplikacjom	26
Fizyczny dostęp nie jest niezbędny	27
Podsumowanie	29
Część I. Hakowanie	31
2. Podstawowe techniki łamania zabezpieczeń systemu iOS	33
Dlaczego należy wiedzieć, jak łamie się zabezpieczenia	33
Zdejmowanie blokady	34
Narzędzia programistyczne	34
Zdejmowanie blokady dla użytkownika	36
Zdejmowanie blokady w iPhone	37
Tryb DFU	38
Tethered jailbreak a untethered jailbreak	40
Usuwanie blokady z urządzeń i wstawianie kodu	40
Budowa własnego kodu	41
Analizowanie pliku binarnego	42
Testowanie pliku binarnego	45
Demonizowanie kodu	46
Wdrażanie złośliwego kodu przy użyciu archiwum tar	49
Wdrażanie złośliwego kodu przy użyciu dysku RAM	50
Ćwiczenia	60
Podsumowanie	61

3. Wykradanie systemu plików	63
Pełne szyfrowanie dysku	63
Pamięć NAND	63
Szyfrowanie dysków	64
Kiedy szyfrowanie dysku systemu iOS zawodzi	66
Kopiowanie systemu plików w czasie jego działania	66
Program DataTheft	67
Dostosowywanie narzędzia launchd	74
Przygotowywanie dysku RAM	78
Tworzenie obrazu systemu plików	79
Kopiowanie surowego systemu plików	80
Program do kopiowania surowego systemu plików	81
Dostosowywanie narzędzia launchd	84
Przygotowywanie dysku RAM	85
Tworzenie obrazu systemu plików	86
Ćwiczenia	87
Rola inżynierii społecznej	87
Wyłączenie podmienionego urządzenia	88
Dezaktywacja podmienionego urządzenia	88
Podrzucenie urządzenia ze złośliwym oprogramowaniem	89
Aplikacja do przechwytywania hasła	90
Podsumowanie	90
4. Ślady pozostawiane przez aplikacje i wycieki danych	93
Wydobywanie geotagów zdjęć	94
Skonsolidowana pamięć podręczna GPS	95
Bazy danych SQLite	96
Łączenie się z bazą danych	97
Polecenia bazy danych SQLite	97
Wydawanie poleceń SQL	98
Ważne pliki bazy danych	99
Kontakty książki adresowej	99
Obrazy graficzne z książki adresowej	101
Dane Map Google	102
Wydarzenia kalendarzowe	106
Historia rozmów	107
Baza danych wiadomości e-mail	108
Notatki	109
Metadane zdjęć	109
Wiadomości SMS	110
Zakładki przeglądarki Safari	110
Pamięć podręczna SMS-ów funkcji Spotlight	111
Pamięci podręczne przeglądarki Safari	111

Pamięć podręczna aplikacji sieciowych	111
Pamięć WebKit	111
Poczta głosowa	111
Inżynieria wsteczna niepełnych pól bazy danych	112
Wersje robocze SMS-ów	113
Listy właściwości	114
Ważne pliki list właściwości	114
Inne ważne pliki	118
Podsumowanie	120
5. Łamanie szyfrów	121
Narzędzia ochrony danych firmy Sogeti	121
Instalowanie narzędzi ochrony danych	122
Kompilowanie narzędzia do wykonywania ataków metodą brute force	122
Kompilowanie potrzebnych bibliotek Pythona	123
Wydobywanie kluczy szyfrowania	124
Program KeyTheft	124
Dostosowywanie programu launchd	125
Przygotowywanie dysku RAM	126
Przygotowywanie jądra	126
Przeprowadzanie ataku metodą brute force	127
Rozszyfrowywanie pęku kluczy	129
Rozszyfrowywanie surowego dysku	131
Rozszyfrowywanie kopii zapasowych z iTunes	132
Łamanie zabezpieczeń przy użyciu programów szpiegujących	133
Program SpyTheft	133
Demonizowanie programu spyd	137
Dostosowywanie launchd	137
Przygotowywanie dysku RAM	138
Uruchamianie programu szpiegującego	139
Ćwiczenia	139
Podsumowanie	139
6. Odzyskiwanie skasowanych plików.....	141
Wydobywanie danych z kroniki HFS	142
Wydobywanie danych z pustej przestrzeni	143
Najczęściej odzyskiwane dane	144
Zrzuty ekranu aplikacji	144
Usunięte listy właściwości	145
Usunięte nagrania i poczta głosowa	145
Usunięta pamięć klawiatury	146
Zdjęcia i inne dane osobiste	146
Podsumowanie	146

7. Szperanie w systemie wykonawczym.....	147
Analizowanie plików binarnych	148
Format Mach-O	148
Podstawy używania narzędzia class-dump-z	151
Tablice symboli	152
Zaszyfrowane pliki binarne	153
Obliczanie pozycji	155
Wykonywanie zrzutu pamięci	156
Kopiowanie rozszyfrowanego kodu z powrotem do pliku	157
Zerowanie wartości cryptid	158
Atakowanie systemu wykonawczego przy użyciu Cypript	160
Instalowanie Cypript	160
Używanie Cypript	161
Łamanie prostych blokad	162
Podmienianie metod	167
Poszukiwanie danych	169
Rejestrowanie danych	171
Poważniejsze konsekwencje	172
Ćwiczenia	178
Animacje SpringBoard	179
Odbieranie rozmów... z przymusu	179
Robienie zrzutów ekranu	180
Podsumowanie	180
8. Hakowanie biblioteki systemu wykonawczego	181
Rozkładanie Objective-C	181
Zmienne egzemplarzowe	183
Metody	183
Pamięć podręczna metod	184
Dezasemblacja i debugowanie	184
Podśluchiwanie	188
Środowisko języka Objective-C	190
Łączenie z Objective-C	191
Wstawianie złośliwego kodu	193
Program CodeTheft	193
Wstawianie kodu za pomocą debugera	194
Wstawianie kodu za pomocą dynamicznego konsolidatora	196
Pełna infekcja urządzenia	197
Podsumowanie	198
9. Przechwytywanie ruchu	199
Przechwytywanie APN	199
Dostarczanie złośliwego kodu	201
Usuwanie	203

Prosta konfiguracja serwera proxy	204
Atakowanie protokołu SSL	204
SSLStrip	204
Paros Proxy	206
Ostrzeżenia przeglądarek internetowych	207
Atakowanie mechanizmu weryfikacji SSL na poziomie aplikacji	209
Program SSLTheft	209
Przechwytywanie klas HTTP z biblioteki Foundation	214
Program POSTTheft	214
Analizowanie danych	216
Driftnet	218
Kompilacja	218
Uruchamianie narzędzia	219
Ćwiczenia	219
Podsumowanie	221
Część II. Zabezpieczanie aplikacji.....	223
10. Implementowanie algorytmów szyfrowania	225
Siła hasła	225
Uwaga na generatory losowych haseł	228
Wprowadzenie do Common Crypto	228
Operacje bezstanowe	229
Szyfrowanie stanowe	232
Szyfrowanie z kluczem głównym	235
Geoszyfrowanie	239
Geoszyfrowanie z hasłem	242
Dzielenie kluczy na serwerze	243
Zabezpieczanie pamięci	245
Czyszczenie pamięci	246
Kryptografia klucza publicznego	247
Ćwiczenia	251
11. Zacieranie śladów	253
Bezpieczne kasowanie plików	253
Kasowanie przy użyciu algorytmu DOD 5220.22-M	254
Objective-C	255
Kasowanie rekordów SQLite	257
Pamięć klawiatury	261
Randomizowanie cyfr kodu PIN	262
Zrzuty widoku okien aplikacji	263

12. Zabezpieczanie systemu wykonawczego.....	265
Reagowanie na próby modyfikacji	265
Kasowanie danych użytkownika	266
Wyłączanie dostępu do sieci	266
Raportowanie do centrali	267
Rejestrowanie zdarzeń	267
Fałszywe kontakty i wyłączniki awaryjne	267
Sprawdzanie danych procesu	268
Blokowanie debuggerów	270
Sprawdzanie integralności klas systemu wykonawczego	272
Sprawdzanie poprawności przestrzeni adresowej	272
Funkcje śródliniowe	281
Utrudnianie dezasemblacji	287
Flagi optymalizacyjne	287
Usuwanie symboli	291
Flaga -funroll-loops	296
Ćwiczenia	299
13. Wykrywanie zdjęcia blokady.....	301
Test integralności piaskownicy	301
Testy systemu plików	303
Obecność plików związanych ze zdejmowaniem blokady	303
Rozmiar pliku /etc/fstab	304
Ślady dowiązań symbolicznych	305
Test wykonywania strony	305
14. Dalszy rozwój.....	307
Myśl jak haker	307
Inne narzędzia do inżynierii wstecznej	307
Bezpieczeństwo a zarządzanie kodem	308
Elastyczne podejście do kwestii bezpieczeństwa	309
Inne wartościowe książki	310
Skorowidz	311

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 zawile 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 wymieniaania 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;
- połączenie 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 hasle. 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:\\"
        " ZXCVCBNM<>? ";

    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 hasle 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 informacji 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, CCAAlgorithm 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.

`CCAAlgorithm 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
Od462b3ec789cfafc50f0bba49cc73507015ac24ec548bd1ef5a45a770eb34985296256a1c0073021b26c
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 kryżowego 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 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 hasel. 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 żadaną 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 tęczyowych 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 sfalszował). 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",
    *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/CommonCrypto.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 sfalszować 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 iPhone'ie 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 m	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 hasel, 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 bufor: 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 podmiiany 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 lokalizację 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 dostępne 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 dostępne 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 zagląda 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;
    CTypeRef keys[2];
    CTypeRef 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

A

adres
 APN, 199
 klasy, 190, 195
 końcowy zaszyfrowanych danych, 157
 MAC, 236
 początkowy zaszyfrowanych danych, 157
alarm, 106
albumy fotograficzne, 119
algorytm
 AES, 63, 174
 AES-128, 230
 DOD 5220.22-M, 254
algorytmy szyfrujące, 228
analizator sieciowy, 206
analizowanie
 danych, 216
 kodu launchd, 57
 plików binarnych, 148
animacje SpringBoard, 179
aplikacja
 AOL Instant Messenger, 216
 klawiaturowa, 262
 Mapy Google, 102
 PayPal, 210
 Settings, 203
 System Profiler, 38
aplikacje
 finansowe, 174
 Objective-C, 147
APN, Access Point Name, 199
Apple Configuration Utility, 199
architektura
 ARM, 42
 armv7, 155, 159
archiwum tar, 41, 49
atak
 0-day, 301
 brute force, 122, 126, 127
 człowiek pośrodku, man-in-the-middle, 25, 207, 247
 Evil Maid Attack, 133
 kryptoanalityczny, 228, 236
 metodą powtórkową, 228

 przy użyciu konfiguracji APN, 204
 słownikowy, 236
atakowanie
 mechanizmu weryfikacji SSL, 209
 protokołu SSL, 204
 zasad zarządzania, 168
atrybut
 always_inline, 286
 cprotect, 141, 253
automatyczne odbieranie połączenia, 179

B

bankowość elektroniczna, 177
baza danych SQLite, 96, 258
bezpieczeństwo, 223, 309
 aplikacji, 18
 danych, 25
 fizyczne, 27
 modułów, 17
 nieużywanych danych, 21
 urządzeń, 18
bezpieczne kasowanie
 danych, 146
 plików, 253
biblioteka
 C SQLite, 260
 Common Crypto, 228
 Foundation, 153, 214, 246
 libgcc, 43
 libplist, 79
 libusb-devel, 79
 Security, 248
biblioteki
 kryptograficzne, 251
 Pythona, 123
blokada GUI, 26, 164
blokowanie
 debugerów, 270
 interfejsu, 91
błąd segmentacji, 270

C

certyfikaty z rekordami parowania, 130
chrootowane środowisko, 74
Common Crypto, 21
Cycrypt, 36, 160–178, 193
Cydia, 49, 303
czas bezwzględny Mac, 107

D

dane
 aplikacji, 164
 konfiguracyjne, 115
 osobiste, 146
 rozszyfrowane, 177
 uwierzytelniające, 214
debuger
 gdb, 184
 GNU Debugger, 154, 194
demon, 46
 launchd, 75
 SpyTheft, 134
 SSH, 304
 usbmux, 79
 usbmuxd, 68
deszyfrowanie, 124, 129
 kopii z iTunes, 132
 plików, 156
 surowego dysku, 131
 tekstu, 231
dezasemblacja, 42, 184, 287
 funkcji main, 185
 programu, 268, 282, 297
DFU, Device Firmware Upgrade, 38
dostarczanie złośliwego kodu, 201
dostęp do
 danych aplikacji, 177
 GUI, 49
 interfejsu, 175
 obiektu klasy, 175
 sieci, 266
dostosowywanie programu launchd, 74, 84, 125, 137
DRM, Digital Rights Management, 22
dynamiczne
 biblioteki, 198
 zależności, 43
dynamiczny konsolidator, 193, 195, 281
dysk RAM, 50, 78, 86, 138

E

ekran kodu PIN, 163
eksploit, 28
eksploit limera1n, 38
element słownika, 175
elementy SecKeyRef, 248

e-mail, 108
entropy, 240

F

falszywe kontakty, 267
FIPS 140-2, 17
firma Sogeti, 121
flaga
 -funroll-loops, 296, 297
 isysroot, 42
 konsolidatora -lobjc, 149, 196
 P_TRACED, 269
 PT_DENY_ATTACH, 270
flagi optymalizacyjne, 287
folder, *Patrz* katalog
format Mach-O, 148
FTL, flash translation layer, 39
funkcja
 CCCrypt, 229
 CCCryptorFinal, 233
 check_debugger, 296
 class_getMethodImplementation, 191
 class_getName, 190
 class_replaceMethod, 195
 dladdr, 272
 dlopen, 195
 dlsym, 171, 195
 encrypt_AES128, 234
 fsexec, 75
 generująca parę kluczy, 244
 geoszyfrowania, 241
 is_session_valid, 284
 isDecryptedCorrectly, 143
 kasująca pliki, 253
 ls, 169
 lstat, 305
 method_getImplementation, 192
 NSLog, 171
 NSSelectorFromString, 153
 objc_msgSend, 181, 186, 192
 object_getClass, 192
 PBKDF2, 237, 240
 PKCS5_PBKDF2_HMAC_SHA1, 237
 ptrace, 270, 271
 puts, 43, 298
 score_passphrase, 228
 skrót, 166
 Spotlight, 111
 szyfrująca, 233
 szyfrująca klucz główny, 239, 242
 UIImageScreenImage(), 180
 vm_protect, 305
funkcje
 derywacyjne kluczy, 235, 240
 obliczania pochodnych, 173
 śródliniowe, 281

G

- generator haseł, 228
- generowanie kluczy, 249
- geoszyfrowanie, geo-encryption, 239, 241
- geoszyfrowanie z hasłem, 242
- geotagowanie, 94
- GPS, 95

H

- haker, 31
- hasła, 23
- hasła Apple, 46
- hasło do punktu dostępowego WiFi, 130
- historia rozmów, 107, 113

I

- identyfikator RPN, 175
- identyfikatory UUID, 116
- implementacja
 - algorytmu szyfrowania, 225
 - zabezpieczeń, 308
- infekcja urządzenia, 197
- informacje
 - ściśle tajne, 254
 - terminala karty kredytowej, 175
- instalowanie
 - Cycript, 160
 - OpenSSH, 45
 - Paros Proxy, 206
- instrukcja
 - jne, 291
 - leal, 291
 - movl, 290
- integralność
 - klas, 272
 - piaskownicy, 301
- interfejs sieciowy, 219
- interpreter
 - powłoki bash, 304
 - Pythona, 123

J

- jailbreak, 37
 - tethered, 40
 - untethered, 40
- jailbreaking, 34
- jądro, 126
- język Cycript, 160
- języki refleksyjne, 27, 147

K

- kafelki map, 102
- kalendarz, 106
- kasowanie
 - danych, 254, 261
 - danych użytkownika, 266
 - pamięci NAND, 141
 - plików, 253
 - rekordów SQLite, 257
- katalog
 - Applications, 93
 - bin, 46
 - driftnet, 218
 - główny, root, 42
 - junk, 142
 - LaunchDaemons, 47
 - LocalStorage, 111
 - maptiles-output, 103
 - Media, 34
 - pair_records, 117
 - python_scripts, 126, 130, 142
 - Thumbnails, 119
 - Thumbs, 119
 - undelete, 142
- KDF, key derivation function, 235
- keylogger, 119
- kiosk, Newsstand, 152
- klasa
 - AccountManager, 178
 - Accounts, 178
 - AppDelegate, 164
 - AppDelegateclass, 163
 - DTPinLockController, 160, 170
 - EMFVolume, 131
 - FileWiper, 255
 - NSData, 248
 - NSFileProtectionNone, 65
 - NSMutableURLRequest, 214
 - NSMutableURLConnection, 216
 - NSMutableURLRequest, 214, 273
 - NSObject, 153
 - NSString, 246
 - NSURLConnection, 209
 - NSURLRequest, 214
 - SaySomething, 153
 - SkipLimitState, 167, 168
 - UIApplication, 162
- klasy ochrony, 65
- klient poczty e-mail, 145
- klucz
 - BAGI, 65
 - Dkey, 21, 65, 81, 123
 - DYLD_INSERT_LIBRARIES, 213
 - EMF!, 65, 81, 123, 128
 - GID, 21
 - GUID, 21
 - originalPrice, 176

- klucze
 - klas, 128
 - klasy ochrony, 65
 - prywatne, 117, 248
 - publiczne, 247
 - sprzętowe, 21, 65
 - szyfrowania, 129
 - szyfrowania pliku, 24, 253
- kod
 - asemblera, 289
 - do zabezpieczeń aplikacji, 175
 - PIN, 23, 164
- kompilacja
 - Driftnet, 219
 - launchd, 125
- kompilator
 - gcc, 232
 - krzyżowy, 70
 - llvm-gcc, 42, 56
- komunikat, 147
 - alloc, 181
 - init, 181
 - makeKeyAndVisible, 164
 - release, 181
 - say, 181
- konfiguracja
 - APN, 200
 - MDM, 168
 - serwera proxy, 204
 - sfalszowana, 202
 - urządzenia, provisioning, 25
- kontrola
 - dostępu, 27
 - wersji kodu źródłowego, 122
- kontroler
 - UIViewController, 164
 - widoku kart, 164, 165
- kopie pamięci klawiatury, 146
- kopiowanie
 - danych użytkownika, 71
 - surowej zawartości dysku, 81
 - systemu plików, 26, 66, 80
- kradzież
 - danych, 25–28, 121
 - systemu plików, 63, 93
 - usług, 167, 168
- kronika HFS, 142, 253
 - listy właściwości, 145
 - usunięte kopie pamięci klawiatury, 146
 - usunięte zdjęcia, 146
 - zrzut ekranu, 144
- kryptografia klucza publicznego, 247
- książka adresowa, 99

L

- lista klas, 171
- lista właściwości, 114, 154

- logowanie się w urzędzeniu, 123
- luki w zabezpieczeniach, 28, 172, 177

Ł

- ładowanie dynamicznych bibliotek, 158
- łamanie
 - kodu PIN, 50, 122
 - prostych blokad, 162
 - szyfrów, 121–140
 - zabezpieczenia ekranu blokującego, 165
 - zabezpieczeń, 33, 133
- łańcuchowanie szyfrowanych bloków, 228
- łączenie Cycript z PhotoVault, 162
- łączenie się z bazą danych, 97

M

- magazyn wymazywalny, 21
- makro printf, 43
- manifest launchd, 46
- Mapy Google, 102
- MDM Apple, 301
- mechanizmy reakcji na modyfikację, 265
- metadane zdjęć, 109
- metoda, 183
 - alloc, 184
 - applicationDidEnterBackground, 264
 - applicationWillResignActive, 264
 - bytes, 246
 - CFStringGetCharactersPtrfunction, 247
 - init, 184, 187
 - initialize, 187
 - pinLockControllerDidFinishUnlocking, 163, 170
 - POST, 214
 - release, 184
 - say, 184, 191
 - setHTTPBody, 214, 273
 - sharedApplication, 192
 - skipsForStation, 168
 - userIsLogged, 191
 - UTF8String, 246
 - writeToFile, 172
- metody
 - dowolnej klasy, 171
 - klasowe, 153
 - szyfrowania, 21
 - zdejmowania blokad, 36, 40
- mieszanie kluczy, 243
- moduł
 - construct, 123
 - kryptograficzny, 17
 - m2crypto, 123
 - pycrypto, 123
- monitorowanie aktywności aplikacji, 34
- montowanie woluminu, 78

N

nagłówek Mach-O, 148
nagrania poczty głosowej, 145
narzędzia
 analityczne, 35
 deszyfrujące, 121
 diagnostyczne, 35
 dla programistów, 36
 do debugowania, 308
 do dezasemblacji, 308
 do monitorowania, 34
 inżynierii wstecznej, 308
 ochrony danych, 121
narzędzie, *Patrz* program
nasłuchiwanie połączeń, 67, 80, 139
notatki, 109

O

obraz
 dysku, 88
 książki adresowej, 101
 systemu plików, 86
ochrona danych, 122
ochrona danych Sogeti, 131
odblokowywanie urządzenia, *Patrz* zdejmowanie
 blokady
odczytywanie z pamięci kodu PIN, 175
odtworacz Pandora, 167
odzyskiwanie
 danych, 24
 list właściwości, 145
 plików, 141, 146
okno programu redsn0w, 38
omijanie zabezpieczeń PIN, 163, 164
opcja auto_vacuum, 259
opcje programu redsn0w, 59
OpenSSH, 45
operacje bezstanowe, 229
oprogramowanie szpiegowskie, 26
ostrzeżenia przeglądark, 207

P

pakiet
 adv-cmds, 160
 Erica Utilities, 154
 mobilesubstrate, 160
 usbmuxd, 79
pamięć
 GPS, 95
 klawiatury, 261
 NAND, 63
pamięć podręczna
 aplikacji sieciowych, 111
 GPS, 95

klawiatury, 93
 metod, 184
 przeglądarki Safari, 111
 SMS-ów, 111
 WebKit, 111
Pandora Radio, 167
Paros Proxy, 206
PBKDF2, password-based key derivation function,
 236
pek kluczy, 129
piaskownica, 172
PKI, public key infrastructure, 204
plik
 AddressBook.txt, 100, 102
 AddressBookImages.sqlitedb, 101
 blank.png, 103, 106
 Bookmarks.db, 110
 Bookmarks.plist.anchor.plist, 116
 cache.plist, 114
 Calendar.sqlitedb, 106
 call_history.db, 107
 CFBundleExecutable, 154
 ch03_tar_binaries.zip, 79
 com.apple.accountsettings.plist, 116
 com.apple.commcenter.plist, 116
 com.apple.conference.history.plist, 116
 com.apple.Maps.plist, 116
 com.apple.mobile.installation.plist, 116, 120
 com.apple.mobilephone.plist, 116
 com.apple.mobilephone.speeddial.plist, 116
 com.apple.mobilesafari.plist, 116
 com.apple.network.identification.plist, 118
 com.apple.preferences.network.plist, 118
 com.apple.UIKit.pboard/pasteboard, 120
 com.apple.wifi.plist, 117
 com.apple.youtube.plist, 116
 com.yourdomain.spyd.plist, 137
 Cookies.binarycookies, 118
 Cydia.app, 303
 cydia.log, 304
 d1cef203c3061030.plist, 131
 data_ark.plist, 117
 dataprotection.log, 127
 DeviceEncryptionKeys.plist, 127, 131
 dynamic-text.dat, 119
 filesystem.tar, 80
 FileWiper.m, 255
 fstab, 304
 HelloWorld.m, 148, 184
 History.plist, 115, 116
 Info.plist, 213
 injection.c, 194, 196
 injection.dylib, 194, 213
 injection.m, 212, 214
 keychain.csv, 131
 key-chain-2.db, 130
 KeyPadViewController.h, 262
 KeyPadViewController.m, 263

- plik
 - KeyTheft.dmg, 126
 - launchd.c, 54, 75
 - LockBackground.cpbitmap, 119
 - merge_maptiles.pl, 104
 - Makefile, 123, 219
 - manifestu launchd, 47
 - MapTiles.sqlitedb, 102
 - MobileSubstrate.dylib, 303
 - notes.sqlite, 109
 - objc.h, 182
 - passphrase_strength.m, 226
 - parse_maptiles.pl, 102
 - payload.c, 81, 124
 - Photos.sqlite, 109
 - png.c, 218
 - proc.h, 269
 - Protected Index, 108, 139
 - ptrace.h, 270
 - runtime.h, 182, 183
 - seccrypt.m, 247
 - securitycheck.c, 282, 284
 - SMS/sms.db, 110
 - spyd.c, 134
 - SpyTheft.dmg, 138
 - sshd, 304
 - sslstrip.log, 205
 - stateful_crypt.m, 233
 - SuspendState.plist, 117
 - syslog, 304
 - TestConnection.m, 210, 277
 - textcrypt.m, 230
 - textdecrypt.m, 231
 - usbmux.c, 68, 81
 - usbmux.o, 84
 - util.c, 123
 - voicemail.db, 112
 - watchdog.c, 67, 81
 - watchdog.o, 84
- pliki
 - .db, 96
 - .emlx, 109
 - .sqlited, 96
 - AMR, 112
 - bazy danych, 99
 - binarne, 45
 - zaszyfrowane, 141
- pobieranie adresu MAC, 236
- podmienianie
 - metod, 167, 195
 - urządzenia, 88, 89
- podpisane pliki binarne, 49
- podpisywanie
 - kluczy, 304
 - plików binarnych, 45, 56
- podśluchiwanie, 188
- polecenia
 - SQLite, 97
 - polecenia załadowania, load commands, 148
- polecenie
 - .dump, 98, 101
 - .exit, 98
 - .headers on, 98
 - .output, 98
 - .schema, 98
 - .tables, 97
 - call, 190
 - cd, 154
 - cmake, 79
 - DELETE, 259
 - hg, 122
 - launchctl, 47
 - make, 219
 - memory dump, 157
 - nc, 80
 - plutil, 154
 - scp, 49, 154
 - strip, 292
 - tail, 171
 - unzip, 154
 - VACUUM, 258
- połączenie gniazdowe, 216
- poszukiwanie danych, 169
- powiadomienie o zdarzeniu, 267
- powtórne użycie klucza, key replay, 228
- poziomy bezpieczeństwa, 17
- proces singletonowy, 174
- program
 - bison, 36
 - blackra1n, 36
 - bruteforce, 122, 142
 - c++filt, 153
 - class-dump-z, 151–153, 158
 - cmake, 79
 - CodeTheft, 193, 196
 - cycrypt, 270
 - Cydia, 49, 50
 - DataTheft, 67, 87, 129
 - Driftnet, 218
 - emf_undelete, 142
 - Exifprobe, 94
 - file, 143
 - gdb, 35
 - gprof, 35
 - hdiutil, 126
 - Hex-Rays De-compiler for ARM, 308
 - IDA Pro, 308
 - ifconfig, 35, 219
 - iproxy, 79, 127
 - iTunes, 79
 - KeyTheft, 129, 142
 - kopiujący dane, 214
 - launchd, 46, 54, 74, 84, 125, 137
 - ldid, 45

- Isof, 35
- MacPorts, 218
- make, 36
- Mercurial, 122
- nc, 80
- netcat, 80, 127, 139, 216
- netstat, 35
- nice, 35
- nm, 35
- oneSafe, 172, 173
- otool, 35, 42, 155, 282
- otools, 149
- patch, 36
- payload, 126
- Photo Vault, 154, 160
- PhotoVault, 165
- POSTTheft, 214
- ps, 35
- RawTheft, 131, 142
- redsn0w, 36–41, 49, 127
- renice, 35
- route, 35
- sn0breeze, 36
- spyd, 137
- SpyTheft, 133
- SSLStrip, 203
- SSLTheft, 209
- strip, 294
- sysctl, 35
- tar, 67
- testowy SSL, 277
- tcpdump, 35
- TestConnection, 211
- protokół
 - SSL, 204
 - UIApplicationDelegate, 162
 - USBMux, 68
- przechowywanie hasła, 174
- przechwytywanie
 - APN, 199
 - danych, 219
 - danych uwiaryzelniających, 217
 - hasła, 90
 - pakietów sieciowych, 199
 - wiadomości, 216
- przedrostek `_OBJC_CLASS_$_`, 153
- przeglądarka Safari, 110
- przekierowanie na serwer proxy, 203
- przekierowanie ruchu, 25
- przerzucanie bitów, bit flipping, 228
- przetwarzanie
 - kafelków, 102
 - płatności, 174
- przygotowywanie jądra, 126
- punkt wstrzymania, 283
- punkty montowania, 78

R

- randomizowanie klawiszy, 262
- refleksja, 149
- rejestr `$r0`, 187
- rejestrwanie danych, 171
 - przychodzących, 216
 - z klawiatury, 119
- rejestrwanie zdarzeń, 267
- rekonstrukcja obrazów z kafelków, 104
- repozytorium apt, 303
- robak SSH, 28
- rozszyfrowywanie pęku kluczy, 129
- rozwijanie
 - pętli, 298
 - procedur, inlining, 269
- RPN, responsible purchasing network, 175
- ruch HTTPS, 205

S

- segmenty Mach-O, 151
- sejfy na dane osobiste, 172
- sekcja danych, 148
- selektor metody, 183
- selektory, 161
- serwer Paros Proxy, 206, 220
- serwer proxy, 206, 214
- sfalszowane profile konfiguracyjne, 203
- sfalszowana konfiguracja APN, 221
- silne hasło, 23, 225
- skrót
 - hasła, 166
 - jednokierunkowy kodu PIN, 166
- skrypt
 - backup4.py, 132
 - emf_decrypter.py, 131
 - emf_undelete.py, 142
 - kernel_patcher.py, 126
 - keychain_tool.py, 130
 - Python, 205
 - sslstrip.py, 205
 - start-server.sh, 206
- skuteczność szyfrowania, 21
- słowo kluczowe static, 287
- SMS, 110
- sól, 236
- sprawdzanie
 - hasel, 226
 - integralności klas, 272
 - przeźrzeni adresowej, 272
- SQL, Structured Query Language, 97
- SQLite, 96
- SQLite Browser, 97
- standard
 - FIPS 140-2, 17
 - RFC 822, 107
- stanowy obiekt szyfrowania, 233

- struktura
 - kontrolera widoku, 176
 - objc_class, 182, 183
 - objc_object, 182
 - SEL, 195
- strumień komunikatów, 168
- surowa zawartość dysku, 81
- symbole, 43
- system
 - plików HFS+, 58
 - wykonawczy aplikacji, 147, 265
 - X11, 218
- systemy
 - zabezpieczeń, 174
 - zasad, 178
- szyfr
 - blokowy, 228
 - strumieniowy, 228
- szyfrowanie, 225
 - 3DES, 21
 - AES, 21, 174
 - AES-Wrap, 141
 - dysku, 63
 - kroniki HFS, 65
 - lokalizacyjne, 240
 - pęku kluczy, 139
 - plików w iOS, 22
 - poufnych danych, 24
 - RC4, 21
 - stanowe, 232
 - systemu plików, 24, 64
 - tekstu, 230
 - torby z kluczami, 66
 - z kluczem głównym, 235

Ś

- ścieżka do
 - dysku RAM, 127
 - plików konfiguracyjnych apt, 304
- ślady dowiązań symbolicznych, 305

T

- tabela
 - ABMultiValue, 100
 - call, 107
 - CellLocation, 95
 - mailboxes, 98
 - message_data, 108
 - WifiLocation, 95
- tablica transferAccounts, 178
- tablice symboli, 152
- technologia DRM, 22
- terminal karty kredytowej, 175
- test
 - integralności metod, 274
 - integralności piaskownicy, 302
 - kolejności bitów, 219
 - połączenia SSL, 210
 - wykonywania strony, 305
- testowanie
 - penetracyjne, 204
 - pliku binarnego, 45
- testy logiczne, 167
- teczowe tablice, 236
- torba na klucze, keybag, 65
- tryb
 - DFU, 38, 40, 49
 - diagnostyczny, 38
- tunelowanie połączeń sieciowych, 67
- tworzenie
 - aplikacji launchd, 50
 - dysku RAM, 50, 58
 - konfiguracji urządzeń, 199
 - obrazu systemu plików, 79, 86
 - plików konfiguracyjnych, 203
 - połączenia USB, 68
 - woluminu UDIF HFS, 78
 - wskaźników na dane w pamięci, 246
- typ danych IMP, 183
- typ symbolu, 44

U

- UDID urządzenia, 236
- układ NAND, 63
- uruchamianie
 - dysku RAM, 59
 - programu szpiegującego, 139
 - wiązane, tethered boot, 139
- USB, 63
- usuwanie
 - plików, 253
 - symboli, 291
- utrata danych, 25, 29
- utrudnianie dezasemblacji, 287
- uwierzytelnianie, 166
- uwierzytelnianie transakcji, 178

W

- wady protokołu SSL, 247
- warstwa
 - Mobile Substrate, 197
 - SSL, 203, 205
 - translacji flash, 39
- wartość
 - cryptoff, 157
 - cryptsize, 157
- wdrażanie
 - archiwum, 49
 - złośliwego kodu, 49, 50
- WebKit, 111

- wektor inicjujący, initialization vector, 228
- wersja robocze SMS-ów, 113
- weryfikacja poprawności deszyfrowania, 177
- weryfikowanie metod klasy, 276
- włamanie do aplikacji, 174
- własność cryptoff, 156
- wolumin
 - HFS, 50
 - HFS UDIF, 58
 - KeyTheft, 126
- wskaznik na strukturę, 195
- wskaźniki, 182
- współrzędne GPS, 94, 175, 242
- wstawianie
 - kodu do klas, 214
 - kodu do manifestu, 216
 - złośliwego kodu, 193–196
 - kodu do programu, 213
- wyciek danych, 93, 174
- wydarzenia kalendarzowe, 107
- wydobywanie danych, 142, 143
- wykrywanie
 - obcych plików, 303
 - zdjęcia blokady, 301
- wyłączanie
 - podpisu, 27
 - weryfikacji SSL, 212, 217
 - zegara, 67
- wyłączniki awaryjne, 268
- wymuszanie przestrzegania zasad, 177
- wywołania systemowe, 51
- wzór na przesunięcie, 158

Z

- zabezpieczanie
 - aplikacji, 20, 308
 - danych, 19
 - danych zdalnych, 25
 - pamięci, 245
 - plików, 128
- zabezpieczenia
 - fabryczne, 18
 - sieciowe, 20
 - urządzeń, 19

- zakładki przeglądarki Safari, 110
- załącznik z konfiguracją urządzenia, 202
- załączniki, 109
- zapisywanie
 - certyfikatów, 131
 - hasel, 131
- zasady zabezpieczeń, 168
- zdalne
 - kasowanie danych, 26
 - wstawianie kodu, 27, 28
 - wykonanie kodu, 28
- zdejmnowanie blokady, jailbreak, 27, 33–40, 121, 301
- zegar mechanizmu bezpieczeństwa, 67
- zerowanie wartości cryptid, 158
- złośliwa dynamiczna biblioteka, 193
- złośliwe oprogramowanie, 33
- złośliwy kod, 49, 201
- zmienna
 - delegate, 162
 - DYLD_INSERT_LIBRARIES, 197, 213
 - inventoryEntry, 176
 - pin, 166
- zmiennne egzemplarzowe, 169, 183
- zrzut
 - ekranu, 120, 144, 180
 - łańcuchów, 44
 - tablicy symboli, 43, 293
- zwrot środków, 176

Ż

- żądanie PT_DENY_ATTACH, 270

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

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!

helion.pl
księgarnia
internetowa

Nr katalogowy: 11712



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900
0 601 339900



Helion

Sprawdź najnowsze promocje:

🔗 <http://helion.pl/promocje>

Książki najchętniej czytane:

🔗 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

🔗 <http://helion.pl/novosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

sięgnij po WIECEJ



KOD KORZYŚCI

ISBN 978-83-246-5147-4



Cena 59,00 zł