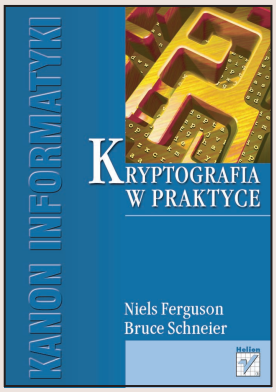
Kryptografia w praktyce



Autorzy: Niels Ferguson, Bruce Schneier
Tłumaczenie: Romasz Żmijewski
ISBN: 83-7361-211-4
Tytuł oryginału: Practical Cryptography
Format: B5, stron: 290

Obecnie najważniejszym zagadnieniem w świecie biznesu jest bezpieczeństwo. Nie mając bezpiecznego systemu komputerowego nie można zarabiać pieniędzy, nie można rozwijać działalności, więc tak naprawdę nie sposób przetrwać na rynku. Kryptografia jawi się jako metoda zapewnienia bezpieczeństwa w cyberprzestrzeni. Co ciekawe, nie pojawiły się jeszcze książki poświęcone implementowaniu kryptografii i włączaniu jej w używane na co dzień systemy.

W większości przypadków kryptografia dała internetowej społeczności niewiele ponad złudne poczucie bezpieczeństwa, gdyż tak naprawdę bezpieczeństwa tego dotąd nie ma. Sytuacja taka nie sprzyja nikomu... poza włamywaczami.

Niniejsza książka, autorstwa spółki niekwestionowanych autorytetów światowych, wypełnia tę lukę pokazując, jak implementować metody kryptografii w praktyce; książka ta stanowi zatem połączenie teorii z praktyką informatyczną.
W książce opisano między innymi:

- Praktyczne zasady doboru i użycia kryptograficznych funkcji elementarnych, od szyfrów blokowych po podpisy cyfrowe.
- Implementację algorytmów kryptograficznych i budowę bezpiecznych systemów.
- Spójną filozofię projektowania dająca gwarancję, że ostatecznie cały system uzyska żądany poziom bezpieczeństwa.
- Dlaczego bezpieczeństwo wpływa na wszystkie składniki systemu i dlaczego ma ono być podstawowym celem projektu?
- Jak proste interfejsy funkcji kryptograficznych pozwalają ograniczyć złożoność systemu i zwiększyć jego bezpieczeństwo?

O autorach:

Niels Ferguson jest inżynierem i konsultantem kryptografii. Ma on ogromne doświadczenie w projektowaniu i implementacji algorytmów i protokołów kryptograficznych oraz dużych systemów zabezpieczeń. Wcześniej pracował na rzecz DigiCash i CWI; w Counterpane Internet Security ściśle współpracował z Brucem Schneierem. Opublikował wiele prac naukowych z dziedziny kryptografii.

Bruce Schneier jest założycielem i dyrektorem technicznym Counterpane Internet Security, firmy zajmującej się monitorowaniem bezpieczeństwa. Ten światowej sławy naukowiec, ekspert w dziedzinie bezpieczeństwa, jest autorem książek „Secrets and Lies: Digital Security in a Networked World" oraz „Applied Cryptography" wydanych przez Wiley Technology Publishing.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

6.

Funkcje mieszające 75

7.

Kody uwierzytelniania wiadomości 85

8.

Bezpieczny kanał 95

16.

17.

18.

19.

20.

21.

22.

9

O implementacji (I)

Czas już powiedzieć co nieco o implementacji. Realizacja systemów kryptograficznych różni się od implementacji zwykłych programów na tyle, że zagadnienie to zasługuje na osobne omówienie.

Najwięcej problemów, jak zwykle, sprawia zasada najsłabszego ogniwa (punkt 2.2). Bardzo łatwo jest zmarnować podczas implementacji cały wysiłek związany z osiągnięciem wymaganego poziomu bezpieczeństwa. To właśnie błędy implementacji (najczęściej w formie przepełnienia bufora) są w praktyce największymi wrogami bezpieczeństwa. Każdy, kto interesował się w ciągu ostatnich kilku lat kwestiami zabezpieczania systemów zrozumie, o co chodzi. W praktyce rzadko zdarza się złamanie systemu kryptograficznego jako takiego. Nie wynika to wcale z doskonałej jakoś systemów; widzieliśmy ich na tyle dużo, by pozbyć się wszelkich złudzeń. Po prostu znacznie łatwiej jest znaleźć błąd w implementacji niż znaleźć słabość kryptograficzną systemu, a atakujący zwykle mają dość rozsądku, by nie męczyć się nad kryptografią, skoro dostępne są metody znacznie prostsze.

Jak dotąd w niniejszej książce zajmowaliśmy się tylko kryptografią, ale tym razem więcej czasu poświęcimy środowisku, w jakim ona funkcjonuje. Każda część systemu wpływa na jego bezpieczeństwo i budując dobry system musimy od początku nie tylko mieć bezpieczeństwo na uwadze, postawić sobie bezpieczeństwo jako podstawowy cel pracy. „System" w sensie, jaki mamy tu na myśli, jest bardzo rozległy. Obejmuje wszystkie elementy, których wadliwe działanie grozi redukcją poziomu bezpieczeństwa.

Jednym z najważniejszych elementów, jak zawsze, jest system operacyjny. Żaden powszechnie dostępny system operacyjny nie powstawał głównie z myślą o bezpieczeństwie. Logiczny będzie zatem wniosek, że nie da się zaimplementować naprawdę bezpiecznego systemu. Nie wiemy, jak można by to zrobić, i nie znamy nikogo, kto by to wiedział. Stosowane w praktyce systemy zawierają wiele elementów, przy których tworzeniu w ogóle nie brano pod uwagę bezpieczeństwa, wobec czego niemożliwe jest uzyskanie takiego poziomu bezpieczeństwa, na jakim naprawdę nam zależy. Czy wobec tego należy się od razu poddać? Oczywiście nie. Tworząc system kryptograficzny, staramy się przynajmniej, by jego część zależna od nas była tak bezpieczna, jak to tylko jest możliwe. Być może tchnie to nieco mentalnością urzędnika: zajmujemy się tylko tym, co dotyczy nas bezpośrednio. Tym niemniej my *naprawdę* troszczymy się także o inne części systemu, ale na tym polu nie mamy zbyt wielkiej swobody. Między innymi właśnie dlatego piszemy tę książkę: chcemy uświadomić innym podstępną naturę bezpieczeństwa i pokazać, jak istotne jest dołożenie wszelkich możliwych starań przy realizacji bezpiecznych systemów.

Jest jeszcze inny ważny powód, dla którego warto zatroszczyć się przynajmniej o poprawność części kryptograficznej: ataki bezpośrednio na elementy kryptograficzne są szczególnie niebezpieczne dlatego, że mogą pozostać niewidoczne, o czym wspominaliśmy już wcześniej. Atakujący,

któremu uda się złamać zabezpieczenia kryptograficzne, prawdopodobnie pozostanie nie zauważony. Można go porównać do włamywacza posiadającego zestaw kluczy do mieszkania: o ile zachowa należytą ostrożność, nikt w ogóle nie zauważy włamania.

Naszym długofalowym celem jest budowa bezpiecznych systemów komputerowych. Aby ten cel osiągnąć, każdy musi zajmować się częścią, za którą jest odpowiedzialny. Nasza praca, o czym traktuje niniejsza książka, polega na opracowaniu skutecznych zabezpieczeń kryptograficznych. Jest oczywiste, że zabezpieczyć będzie trzeba także inne części systemu. Nie wiemy, jak to zrobić, ale być może inni to wiedzą, a być może ktoś tego dopiero się nauczy. Do tego czasu całkowite bezpieczeństwo systemu będzie ograniczone przez jego najsłabsze ogniwo, zaś my dołożymy wszelkich starań, by tym ogniwem nie okazała się kryptografia.

O poprawność elementów kryptograficznych warto zadbać także dlatego, że po zaimplementowaniu systemu wszelkie zmiany są bardzo kłopotliwe. System operacyjny działa na pojedynczym komputerze. Systemy kryptograficzne często są używane w ramach protokołów komunikacyjnych łączących szereg komputerów. Aktualizacje systemu operacyjnego pojedynczego komputera są dość proste i w praktyce często mają miejsce. Modyfikacja sieciowego protokołu komunikacyjnego jest koszmarem. Dlatego właśnie wiele sieci działa do dziś zgodnie z projektami z lat 70. i 80. Musimy pamiętać, że każdy nowo tworzony dziś system kryptograficzny, o ile zostanie przyjęty do powszechnego użytku, prawdopodobnie będzie działał jeszcze przez 30 czy 50 lat. Mamy nadzieję, że przez ten czas pozostałe części systemu staną się znacznie bardziej bezpieczne.

9.1. Tworzenie poprawnych programów

Podstawowy problem związany z implementacją systemów polega na tym, że w informatyce nie wiadomo, jak napisać poprawny program czy moduł (przez „poprawny" rozumiemy taki program, który zawsze zachowuje się zgodnie ze specyfikacją). Trudności pisania poprawnych programów wynikają z kilku przyczyn.

9.1.1. Specyfikacje

Pierwszy problem bierze się stąd, że mało który program ma jasno określone wymagania. Bez specyfikacji nie sposób nawet sprawdzić, czy program jest poprawny. W przypadku takich programów kwestia poprawności jest w ogóle nierozstrzygalna.

Wiele projektów informatycznych ma dokument nazywany specyfikacją funkcjonalną. Teoretycznie powinna być to specyfikacja programu, ale w praktyce dokument ten często nie istnieje, jest niekompletny lub opisuje elementy nie odnoszące się do oczekiwanego zachowania programu. Jak długo brak porządnej specyfikacji, nie może być owy o poprawnym programie.

Istota specyfikacji obejmuje trzy etapy:

- **Wymagania**. Wymagania obejmują nieformalny opis efektów działania programu. Jest to dokument typu „co można za pomocą tego programu zrobić" a nie „jak można coś zrobić". Wymagania często bywają niedookreślone i koncentrują się na całościowym obrazie, z pominięciem szczegółów.
- **Specyfikacja funkcjonalna**. Opis wymagań funkcjonalnych mówi, jak program ma działać. Specyfikacja funkcjonalna obejmuje tylko te elementy, które da się sprawdzić nie zaglądając do wnętrza programu. W przypadku każdej pozycji specyfikacji funkcjonalnej należy zadać sobie pytanie, czy możliwe jest przeprowadzenie testu gotowego programu, który byłby w stanie

rozstrzygnąć o spełnieniu danego wymogu. Taki test może badać jedynie zewnętrzne zachowanie się programu, nie może natomiast analizować żadnych stanów wewnętrznych. Żaden wymóg, dla którego nie da się stworzyć odpowiedniego testu, nie może należeć do specyfikacji funkcjonalnej.

Opis wymagań funkcjonalnych powinien być zupełny. Oznacza to, że powinien on obejmować wszystkie elementy funkcjonalne. Żaden element nie ujęty w specyfikacji nie musi być implementowany.

Inny sposób patrzenia na specyfikację funkcjonalną polega na testowaniu gotowego programu. Każdy wymóg może być i powinien być przetestowany.

- **Projekt implementacji**. Dokument ten miewa rozmaite nazwy, ale opisuje on sposób działania programu od wewnątrz. Projekt zawiera wszystko, czego nie można przetestować z zewnątrz. Dobry projekt zwykle opisuje podział programu na moduły oraz funkcjonalność tych modułów. Z kolei na opisy modułów można patrzeć jak na wymagania wobec modułu. W tej sytuacji można z kolei powtórzyć cały cykl na poziomie modułów.

Spośród wskazanych trzech dokumentów najważniejsza jest niewątpliwie specyfikacja funkcjonalna. To według tego dokumentu będzie przebiegać testowanie gotowego programu. Czasami można obejść się bez nieformalnego opisu wymagań, a za projekt starczy kilka szkiców na tablicy. Jednak bez specyfikacji funkcjonalnej nie sposób nawet opisać zakresu prac ani zdecydować, czy ich cel został zrealizowany.

9.1.2. Testowanie i poprawki

Drugi problem związany z pisaniem poprawnych programów dotyczy powszechnie praktykowanego cyklu testowania i robienia poprawek. Programista pisze program i sprawdza, czy zachowuje się on zgodnie z oczekiwaniami. Jeśli nie, to poprawia błędy i ponawia testowanie. Sposób ten, jak wiadomo, nie prowadzi do uzyskania poprawnego programu. Wynikiem jest program, który działa w typowych sytuacjach.

W roku 1972 Edsger Dijkstra w swoim artykule (za który otrzymał Nagrodę Turinga) wyraził myśl, że testowanie może ujawnić jedynie błędy, nie może zaś zagwarantować braku błędów [23]. Co do prawdziwości tego stwierdzenia nie ma żadnych wątpliwości, a my chcielibyśmy pisać programy, których poprawności można byłoby dowieść. Niestety, istniejące obecnie techniki dowodzenia poprawności programów nie nadają się do zastosowania nawet w rutynowej pracy programistycznej, nie mówiąc już o całych projektach.

W obecnej chwili teoretycy informatyki nie znają odpowiedniego rozwiązania problemu poprawności. Być może w przyszłości udowodnienie poprawności programu będzie możliwe. Być może należy wypracować też narzędzia i metodykę do znacznie bardziej dokładnego i wszechstronnego testowanie. Jednak nawet nie dysponując pełnym rozwiązaniem możemy dokładać wszelkich starań i używać wszystkich dostępnych obecnie narzędzi.

Jest kilka dziecinnie prostych zasad dotyczących błędów. Można je znaleźć w każdym podręczniku inżynierii oprogramowania:

- Przy poszukiwaniu błędów najpierw należy zrealizować test, który jest w stanie wykryć dany typ błędu. Następnie trzeba sprawdzić, czy błąd faktycznie zostanie wykryty. Wtedy należy błąd poprawić i ponownie przeprowadzić test sprawdzając, czy błąd już nie wystąpi. Ten sam test musi być wykonywany we wszystkich kolejnych wersjach, aby zagwarantować, że błąd nie pojawi się ponownie.

- Po znalezieniu błędu zawsze należy zastanowić się nad przyczyną jego wystąpienia. Czy w innych częściach programu może się zdarzyć podobny błąd? Należy sprawdzić wszystkie podejrzane miejsca.
- Należy rejestrować wszystkie znalezione błędy. Prosta analiza statystyczna znalezionych błędów może wskazać, które części programu mają szczególnie dużo niedoróbek, jakiego typu błędy pojawiają się najczęściej i tak dalej. Tego typu informacje zwrotne są niezbędne dla kontroli jakości.

Zasady te nie stanowią nawet niezbędnego minimum, ale z drugiej strony niewiele jest gotowych opracowań, z których można byłoby korzystać. Jakości oprogramowania poświęconych jest niewiele książek. Na dodatek poglądy ich autorów nie są ze sobą zgodne. Wielu autorów opisuje konkretną metodologię tworzenia oprogramowania jako jedyne słuszne rozwiązanie, a wszelkie panacea są dla nas podejrzane. Prawda zawsze leży pośrodku.

9.1.3. Lekceważące podejście

Trzecim problemem jest niesłychanie lekceważący stosunek większości informatyków do błędów. Błędy w programach są po prostu traktowane jako fakt naturalny. Sytuacja, w której procesor tekstu nagle przestaje działać powodując utratę pracy z całego dnia, bywa traktowana jak normalny stan rzeczy. Często winę przenosi się na użytkownika: „powinieneś częściej zapisywać swoją pracę". Do powszechnej praktyki firm *software*-owych należy sprzedaż produktów zawierających nieznane błędy. Nie byłby to wielki problem, gdyby chodziło o gry komputerowe, ale w obecnych czasach nasza praca, ekonomia i w coraz większym stopniu całe życie zależą od oprogramowania. Producent samochodów, który znajdzie usterkę (błąd) w sprzedanym samochodzie, ujawnia ją i zobowiązuje się do jej usunięcia. Firmy informatyczne zrzucają z siebie wszelką odpowiedzialność odpowiednio formułując umowy licencyjne — podobne warunki byłyby nie do przyjęcia w przypadku innych produktów. Przy takiej postawie nikt nie będzie dokładać starań, by jego oprogramowanie było poprawne.

9.1.4. Co zatem robić?

W żadnym wypadku nie należy się spodziewać, że do produkcji poprawnego oprogramowania wystarczy dobry programista, uważne sprawdzanie kodu źródłowego, certyfikat ISO 9001 czy dokładne testowanie, a nawet kombinacja wszystkich tych elementów. W praktyce rzecz jest znacznie trudniejsza. Oprogramowanie jest produktem zbyt skomplikowanym, by dało się nad nim zapanować za pomocą kilku zasad i procedur. Lepiej wziąć wzór z najlepszego na świecie inżynierskiego systemu kontroli jakości: chodzi o przemysł lotniczy. Wszyscy uczestnicy rynku lotniczego zaangażowani są w system bezpieczeństwa. Rygorystyczne zasady i procedury dotyczą niemal wszystkich działań. Istnieje mnóstwo wariantów awaryjnych. Każda nakrętka i każda śrubka w samolocie muszą być zatwierdzone do użycia w lotnictwie, zanim można je będzie gdziekolwiek zamontować. Każde podejście mechanika ze śrubokrętem w ręku do samolotu jest nadzorowane i potwierdzane przez innego pracownika. Każdą modyfikację starannie się dokumentuje. Wszystkie wypadki są drobiazgowo badane, aby znaleźć i usunąć ich faktyczne przyczyny. To fanatyczne wprost dążenie do wysokiej jakości jest niezwykle kosztowne. Samolot jest prawdopodobnie o rząd wielkości droższy niż mógłby być, gdyby jego plany konstrukcyjne zostały przesłane do zwykłej firmy

produkcyjnej. Jednak z drugiej strony to dążenie do jakości daje niesamowite efekty. Latanie jest dzisiaj czynnością rutynową, mimo że każda usterka w samolocie może okazać się krytyczna. Pilot w razie kłopotów nie może po prostu wcisnąć hamulców i zatrzymać maszyny. Jedynym sposobem bezpiecznego powrotu jest bardzo delikatna operacja lądowania; niewiele jest na świecie miejsc, w których można ją bezpiecznie przeprowadzić. Przemysł lotniczy zadziwiająco skutecznie zatroszczył się o bezpieczeństwo lotów. Byłoby wskazane przejąć możliwie wiele z wypracowanych przezeń procedur. Być może napisanie poprawnego oprogramowania *musi* być o rząd wielkości droższe, niż jest obecnie, kiedy programy pisze się tak, jak do tego przywykliśmy. Jeśli jednak wziąć pod uwagę koszty społeczne błędów oprogramowania, możemy być pewni, że w dłuższym okresie czasu wysiłek się opłaci.

9.2. Tworzenie bezpiecznego oprogramowania

Jak dotąd mówiliśmy jedynie o poprawności oprogramowania. Samo tylko pisanie poprawnych programów nie wystarcza do stworzenia bezpiecznego systemu. Oprogramowanie musi być nie tylko poprawne, ale także bezpieczne.

Na czym polega różnica? Oprogramowanie poprawne realizuje pewną opisaną specyfikację: wciśnięcie przycisku *A* wywoła skutek *B*. Wobec bezpiecznego oprogramowania stawia się jeszcze jeden wymóg: specyfikację negatywną, polegającą na *niemożności* zrealizowania pewnych funkcji. Niezależnie od tego, co zrobi atakujący, nie może zdarzyć się sytuacja *X*. Jest to zasadnicza różnica: testować można jedynie funkcjonalność, ale nie jej brak. Aspektów związanych z bezpieczeństwem nie można skutecznie przetestować, przez co tworzenie oprogramowania bezpiecznego jest znacznie trudniejsze od tworzenia oprogramowania poprawnego. Nieuchronny jest następujący wniosek:

Standardowe techniki programowania są całkowicie nieprzydatne przy tworzeniu bezpiecznych programów.

Tak naprawdę nie wiemy, jak tworzyć kod źródłowy bezpiecznego programu. Jakość oprogramowania jest tak szerokim zagadnieniem, że należałoby jej poświęcić kilka książek. Nasza wiedza nie wystarcza co prawda do ich napisania, ale *znamy* zagadnienia specyficzne dla kryptografii oraz wiemy, jakie problemy pojawiają się najczęściej. Tymi właśnie problemami zajmiemy się do końca bieżącego rozdziału.

Zanim zaczniemy, sformułujemy jasno nasz punkt widzenia: nikomu, kto nie zamierza włożyć solidnej pracy w stworzenie bezpiecznego programu, nie warto w ogóle kłopotać się kryptografią. Tworzenie systemów kryptograficznych może być dobrą zabawą, ale w przypadku błędnej implementacji jest zwykłą stratą czasu.

9.3. Zachowywanie tajemnic

Każde zastosowanie metod kryptografii wiąże się z istnieniem tajemnic, które muszą pozostać poufne. Oznacza to, że oprogramowanie obsługujące tajemnice musi dopilnować, by nie wydostały się one na zewnątrz.

W przypadku bezpiecznego kanału mamy do czynienia z dwoma rodzajami tajemnic: z kluczami i z danymi. Obie te tajemnice są zresztą tajemnicami chwilowymi: nie zamierzamy przechowywać ich zbyt długo. Dane są potrzebne tylko w chwili przetwarzania wiadomości. Klucze są używane

tylko przez czas istnienia bezpiecznego kanału. Długoterminowe przechowywanie tajemnic zostanie omówione w rozdziale 22.

Tajemnice tymczasowe przechowuje się w pamięci. Niestety, pamięć większości komputerów nie jest miejscem zbyt bezpiecznym. Po kolei omówimy wszystkie typowe problemy z nią związane.

9.3.1. Kasowanie pamięci stanu

Podstawowa zasada związana z tworzeniem oprogramowania dla celów bezpieczeństwa każe usuwać z pamięci wszelkie informacje natychmiast, kiedy tylko przestają być potrzebne. Im dłużej takie informacje są przechowywane, tym większe prawdopodobieństwo, że komuś uda się do nich sięgnąć. Co więcej, ostateczne usunięcie danych powinno nastąpić jeszcze przed utratą kontroli nad nośnikiem, na którym dane te zostały zapisane. W przypadku tajemnic tymczasowych chodzi o czyszczenie odpowiednich fragmentów pamięci.

Mimo że sama zasada wydaje się prosta, rodzi ona zadziwiająco wiele problemów. W przypadku programu pisanego w języku C można zadbać samemu o wyczyszczenie pamięci. Pisząc bibliotekę przeznaczoną dla innych programów trzeba przyjąć, że program główny wywoła odpowiednie procedury, kiedy przestanie potrzebować danych. Na przykład przed zamknięciem połączenia komunikacyjnego biblioteka kryptograficzna powinna się dowiedzieć o tym zamiarze, aby mogła usunąć z pamięci stan sesji bezpiecznego kanału. Biblioteka powinna zawierać odpowiednią funkcję, ale musimy uświadomić sobie, że programista może być zbyt leniwy, by jej użyć. W końcu nawet bez jej wywołania program i tak działa znakomicie.

W niektórych językach obiektowych sytuacja nieco się upraszcza. W C++ każdy obiekt posiada swój destruktor, i fakt ten można wykorzystać do skasowania stanu. Taka praktyka jest standardem w programowaniu procedur obsługi bezpieczeństwa w C++. O ile tylko program główny zachowuje się prawidłowo i niszczy wszystkie obiekty, kiedy przestają być potrzebne, pamięć opisująca stan jest prawidłowo wymazywana. Język C++ zapewnia, że obiekty alokowane na stosie będą niszczone w miarę opróżniania stosu w trakcie obsługi wyjątków, ale o zniszczenie obiektów alokowanych na stercie musi zadbać sam program. Wywołanie funkcji systemu operacyjnego dla zakończenia działania programu może nawet nie opróżnić stosu, a my musimy zapewnić, że nawet w takiej sytuacji dane zostaną wymazane. W końcu system operacyjny wcale nie gwarantuje, że skasuje stan pamięci przed przekazaniem sterowania następnej aplikacji.

Nawet jeśli zrobimy wszystko co trzeba, komputer nadal może odmawiać posłuszeństwa. Niektóre kompilatory starają się zbyt wiele optymalizować. Typowe funkcje związane z bezpieczeństwem część obliczeń wykonują na zmiennych lokalnych, a potem starają się te zmienne usunąć. W języku C można to zrobić wywołując funkcję `memset`. Dobry kompilator zoptymalizuje funkcję `memset` zastępując ją szybszym kodem wstawianym (*in-line*). Jednak niektóre kompilatory domyślają się zbyt wiele. Wykrywają, że zamazywana zmienna lub tablica nie będzie już używana i „optymalizują" program wyrzucając wywołanie `memset`. Całość działa szybciej, ale program przestaje się zachowywać zgodnie z naszymi wymaganiami. Nie jest trudno znaleźć program, który odczyta dane znajdujące się akurat w pamięci. Przekazanie niewymazanej pamięci innej bibliotece grozi więc ujawnieniem danych atakującemu. Wobec tego zachodzi konieczność sprawdzania kodu generowanego przez kompilator dla pewności, że faktycznie usuwa on sekrety z pamięci.

W językach takich jak Java sytuacja komplikuje się jeszcze bardziej. Wszystkie obiekty alokowane są na stercie, ze sterty też usuwane są zbędne już dane. Oznacza to, że funkcja finalizująca (podobna do destruktora C++) nie jest wywoływana, dopóki procedura porządkująca pamięć nie stwierdzi, że dany obiekt nie jest już używany. Nie jest nigdzie powiedziane, jak często procedura ta ma być wywoływana i nietrudno sobie wyobrazić sytuację, w której tajne dane będą przebywać

w pamięci przez długi czas. Użycie procedur obsługi wyjątków utrudnia ręczne wymazywanie pamięci. Po wygenerowaniu wyjątku ze stosu zdejmowane są kolejne wywołania, a programista nie ma żadnego wpływu na przebieg akcji, nie może wywołać żadnego swojego kodu; może co najwyżej każdą funkcję umieścić w dużej klauzuli `try`. Jest to rozwiązanie całkiem niepraktyczne, na dodatek należałoby je stosować w całym programie, wobec czego niemożliwe byłoby stworzenie prawidłowo zachowującej się biblioteki Javy do obsługi bezpieczeństwa. Podczas obsługi wyjątków Java zdejmuje dane ze stosu usuwając wszelkie odwołania do obiektów, ale nie usuwa przy tym samych obiektów. Pod tym względem Java zachowuje się naprawdę źle. Najlepsze rozwiązanie, do jakiego doszliśmy, polegało na zapewnieniu wykonania procedur finalizacyjnych przy zamykaniu programu. Polega ono na tym, że metoda `main` programu wykorzystuje instrukcję `try-finally`. Blok `finally` zawiera kod wymuszający wywołanie procedury czyszczenia pamięci, zaś procedura ta stara się wywołać wszystkie metody finalizujące (szczegóły znajdują się w kodzie źródłowym funkcji `System.gc()` i `System.runFinalization()`). Nadal nie mamy gwarancji, że metody finalizujące zostaną wykonane, ale nic więcej nie udało nam się uzyskać. Najlepszym rozwiązaniem byłoby wsparcie od samego języka programowania. W C++ co prawda teoretycznie możliwe jest napisanie programu czyszczącego stan wszystkich zmiennych z chwilą, kiedy tylko przestają być potrzebne, ale wiele innych cech tego języka powoduje, że nie specjalnie się nadaje do tworzenia oprogramowania związanego z bezpieczeństwem. W Javie usuwanie stanu pamięci jest bardzo trudne. Jedno z możliwych udoskonaleń mogłoby polegać na deklarowaniu zmiennych jako „wrażliwych" i na takiej implementacji, która gwarantowałaby ich wymazanie. Jeszcze lepiej byłoby mieć język programowania wymazujący i usuwający wszystkie zbędne już dane. Pozwoliłoby to uniknąć wielu błędów bez znaczącej utraty wydajności.

Tajne dane mogą gromadzić się także w innych miejscach. Wszystkie dane w razie potrzeby są ładowane do rejestrów CPU. Większość języków nie pozwala na czyszczenie rejestrów, ale w komputerach o niewielkiej liczbie rejestrów bardzo małe jest prawdopodobieństwo, że dane pozostaną w nich na dłużej.

W trakcie przełączania kontekstu (kiedy system operacyjny przełącza się z jednego programu na inny) wartości rejestrów CPU są zapisywane w pewnym obszarze pamięci i na ogół pozostają w nim na dłużej. O ile nam wiadomo, nie ma na to żadnej rady, poza możliwością zwiększającej bezpieczeństwo poprawki w systemie operacyjnym.

9.3.2. Plik wymiany

Większość systemów operacyjnych (w tym bieżące wersje Windows oraz wszystkie systemy Unix) korzysta z pamięci wirtualnej w celu zwiększenia liczby równolegle działających programów. Podczas działania programu nie wszystkie jego dane znajdują się w pamięci; część z nich jest przechowywana w pliku wymiany. Kiedy program potrzebuje danych, których akurat nie ma w pamięci, jego wykonywanie jest przerywane. System obsługi pamięci wirtualnej pobiera potrzebne dane z pliku wymiany, a wtedy program może kontynuować swoje działanie. Co więcej, w razie, kiedy system pamięci wirtualnej stwierdzi, że potrzebuje więcej wolnej pamięci, dowolny fragment pamięci należącej do programu może zostać zapisany w pliku wymiany.

Oczywiście większość systemów pamięci wirtualnej nawet nie próbuje ukrywać danych ani szyfrować ich przed zapisaniem na dysk. Większość oprogramowania powstaje z myślą o współpracy z innymi programami we wspólnym środowisku, a nie o pracy we wrogim środowisku, z jakim mamy do czynienia w kryptografii. Tak więc mamy do czynienia z zagrożeniem, które opiszemy następująco: system pamięci wirtualnej może pobrać fragment pamięci naszego programu i zapisać go na dysku. Program użytkowy nigdy nie jest informowany o takiej sytuacji, więc w ogóle nie

jest w stanie na nią zareagować. Załóżmy, że na dysk przeniesiony zostanie fragment pamięci zawierający klucze. W razie awarii komputera lub choćby wyłączenia zasilania dane pozostaną na dysku. Większość systemów operacyjnych pozostawia dane na dyskach nawet w przypadku poprawnego zamknięcia systemu. Zwykle nie ma sposobu na wymazanie pliku wymiany, więc dane mogą leżeć na dysku w nieskończoność. Nikt nie wie, kto w przyszłości uzyska dostęp do takiego pliku wymiany. Dlatego nie możemy sobie pozwolić na ryzyko zapisania naszych tajemnic w pliku wymiany[1].

Jak zatem uniemożliwić systemowi pamięci wirtualnej zapisywanie naszych danych na dysku? W niektórych systemach operacyjnych dostępne są funkcje systemowe informujące system pamięci wirtualnej, że pewnych obszarów pamięci nie wolno przenosić na dysk. Rzadko zdarza się system operacyjny zawierający obsługę bezpiecznego systemu wymiany, w którym zapisywane na dysku dane byłyby chronione kryptograficznie. Jeżeli w naszym systemie nie jest dostępny żaden z opisanych przed chwilą mechanizmów, to mamy pecha. Możemy wtedy tylko głośno ponarzekać na system operacyjny i zrobić wszystko, co da się zrobić w danej sytuacji.

Załóżmy teraz, że mamy środki, by uniemożliwić zapisywanie części pamięci w pliku wymiany. Którego fragmentu pamięci winna dotyczyć taka blokada? Oczywiście należałoby nią objąć wszystkie te fragmenty pamięci, które mogą zawierać poufne dane. Mamy więc następny problem: w wielu środowiskach programowania bardzo trudno jest ustalić, gdzie dokładnie dane zostały umieszczone. Obiekty są często alokowane na stercie, dane globalne można alokować statycznie, zaś zmienne lokalne zazwyczaj umieszcza się na stosie. Ustalanie takich szczegółów jest bardzo skomplikowane i może być źródłem błędów. Prawdopodobnie najlepsze rozwiązanie polega na blokowaniu całej pamięci naszej aplikacji. Jednak nawet ono nie jest tak proste, jak mogłoby się wydawać, gdyż możemy przy okazji przegapić szereg usług systemu operacyjnego, takich jak automatycznie alokowany stos. Poza tym blokowanie pamięci powoduje, że system pamięci wirtualnej staje się nieskuteczny.

Cała rzecz jest o wiele bardziej skomplikowana, niż powinna. Prawidłowe rozwiązanie polega oczywiście na zbudowaniu systemu pamięci wirtualnej, który chroniłby poufność danych. Wiąże się to ze zmianą systemu operacyjnego i wobec tego jest poza naszą kontrolą. Nawet jeśli stosowne funkcje pojawią się w następnej wersji używanego systemu operacyjnego, to i tak trzeba będzie starannie sprawdzić, czy system pamięci wirtualnej dobrze strzeże powierzanych mu tajnych danych.

9.3.3. Pamięć podręczna

We współczesnych komputerach nie mamy do czynienia z jednym tylko rodzajem pamięci. Istnieje cała hierarchia pamięci. Na samym dole jest pamięć główna — o pojemności idącej często w setki megabajtów. Ponieważ pamięć główna jest względnie powolna, stosuje się pamięć podręczną (*cache*). Jest to pamięć mniej obszerna, ale za to szybsza. Zawiera ona kopię ostatnio używanych danych z pamięci głównej. Jednostka centralna, potrzebując danych, najpierw szuka ich w pamięci podręcznej. Jeśli dane tam są, CPU otrzymuje je dość szybko. Jeśli danych w pamięci podręcznej nie ma, to odczytuje je ze stosunkowo powolnej pamięci głównej i kopiuje do pamięci podręcznej na przyszłość. Miejsce w pamięci podręcznej uzyskuje się, wyrzucając z niej jakiś inny fragment danych.

Opisany mechanizm jest istotny, gdyż pamięć podręczna zawiera skopiowane dane, w tym kopie naszych danych poufnych. Problem polega na tym, że kiedy staramy się nasze tajne dane usunąć, usuwanie takie może się nie powieść. W niektórych systemach modyfikacje są zapisywane jedynie w pamięci podręcznej, a nie w pamięci głównej. Dane zostaną później zapisane do pamięci

[1] Nigdy nie powinniśmy zapisywać tajemnic na żadnym trwałym nośniku bez ich uprzedniego zaszyfrowania. Zagadnieniem tym zajmiemy się później.

głównej, ale dopiero wtedy, kiedy w pamięci podręcznej braknie miejsca na jakieś inne dane. Nie znamy wszystkich szczegółów tego mechanizmu, poza tym jest on zależny od typu procesora. Nie da się stwierdzić, czy istnieje jakieś oddziaływanie między modułem alokacji pamięci a systemem pamięci podręcznej, które pozwoliłoby pominąć etap buforowania danych w pamięci podręcznej podczas zwalniania pamięci. Producenci nie wskazują sposobu usuwania danych z pamięci, a żaden mechanizm, który nie jest oficjalnie udokumentowany, nie jest też godny zaufania.

Kolejne niebezpieczeństwo związane z pamięcią podręczną polega na tym, że w pewnych warunkach pamięć taka może otrzymać sygnał o modyfikacji pewnego obszaru pamięci głównej, na przykład przez inny procesor w systemie wieloprocesorowym. Dane w pamięci podręcznej otrzymują status „niepoprawnych", ale zwykle nie oznacza to wcale ich usunięcia. Znowu może się okazać, że gdzieś istnieje kopia naszych danych poufnych.

Na opisane wyżej problemy nie ma ogólnej rady. Nie stanowią też na ogół wielkiego zagrożenia, ponieważ w większości systemów jedynie sam system operacyjny ma bezpośredni dostęp do mechanizmu pamięci podręcznej. Systemowi operacyjnemu tak czy inaczej trzeba ufać, więc i my mu zaufamy. Tym niemniej o opisanych niebezpieczeństwach trzeba pamiętać podczas projektowania, gdyż sprawiają one, że tworzone systemy nie są tak bezpieczne, jak być powinny.

9.3.4. Zatrzymanie danych w pamięci

Wiele osób zaskakuje fakt, że zwykłe nadpisywanie danych w pamięci wcale nie usuwa starych danych. Szczegóły samego procesu zależą od rodzaju używanej pamięci, ale w zasadzie proces zapisywania sprawia, że w odpowiednim miejscu pamięci dane te dopiero zaczynają się pojawiać. Na przykład w razie wyłączenia maszyny stara wartość częściowo może pozostać w pamięci. Wszystko zależy od konkretnych warunków, ale czasem chwilowe wyłączenie i ponowne włączenie zasilania pamięci mogą przyczynić się do ujawnienia starych danych. Inne typy pamięci mogą „przypomnieć sobie" stare dane w przypadku użycia (często nieudokumentowanych) trybów testowania [39].

Zjawiskiem ty kierują różne mechanizmy. Jeżeli będziemy przechowywać pewne dane przez dłuższy czas w tym samym miejscu pamięci typu SRAM, staną się one preferowanym stanem początkowym pamięci po włączeniu zasilania. Wiele lat temu jeden z naszych przyjaciół zetknął się z tym problemem w swoim komputerze domowej produkcji [9]. Napisał on BIOS, w którym użył pewnej magicznej wartości w konkretnym miejscu pamięci; pełniła ona rolę znacznika wskazującego, czy restart był związany ze startem zimnym, czy gorącym[2]. Po pewnym czasie komputer nie chciał się uruchamiać po włączeniu zasilania, gdyż w pamięci zakodowała się na trwałe magiczna wartość, wobec czego każde uruchamianie komputera było traktowane jako restart gorący. Wartości startowe nie były inicjalizowane prawidłowo, więc proces uruchamiania komputera nie udawał się. W tym przypadku rozwiązaniem okazała się wymiana niektórych układów pamięci, przez co w z odpowiedniego miejsca znikła zapamiętana magiczna wartość. Była to dla nas ważna lekcja: okazało się, że pamięć przechowuje więcej danych, niż się na pozór wydaje. Podobny proces ma miejsce w pamięciach DRAM, tyle że w tym wypadku jest on nieco bardziej skomplikowany. DRAM przechowuje mały ładunek na miniaturowych kondensatorach. Materiał izolacyjny wokół kondensatora jest poddawany wpływowi tak tworzonego pola. Wpływ ten powoduje zmianę struktury

[2] W owych czasach budowane w domu komputery programowano wprowadzając bezpośrednio binarny kod języka maszynowego. Powodowało to wiele błędów i jedyną pewną metodą odzyskania panowania nad komputerem po awarii programu było wyzerowanie maszyny. Zimny start następował po wyłączeniu zasilania. Gorący start był wykonywany po wciśnięciu klawisza zerującego. Start gorący nie obejmował inicjacji części stanu maszyny, wobec czego nie kasował ustawień dokonanych wcześniej przez przez użytkownika.

materiału, a ściślej migrację domieszek [39]. Atakujący, mając fizyczny dostęp do pamięci, mógłby teoretycznie odczytać tak zapisane dane.

Istotność opisanego typu zagrożenia jest dyskusyjna; naszym zdaniem jest to zagrożenie realne. Nie chcielibyśmy przecież, by ewentualnej kradzieży komputera towarzyszyła kradzież skasowanych na nim wcześniej danych. Groźby takiej unikniemy tylko troszcząc się samemu, by komputer rzeczywiście „zapominał" wszystkie kasowane dane.

Potrafimy wskazać jedynie częściowe rozwiązanie, które działa przy pewnych założeniach dotyczących pamięci. Rozwiązanie to, nazwane Boojum[3], nadaje się do wymazywania niewielkich porcji danych, na przykład kluczy. Oznaczmy symbolem m dane, które chcemy przechować. Zamiast zapisywać m, wygenerujemy losowy łańcuch R i zapiszemy R oraz $R \oplus m$. Wartości te umieścimy w dwóch różnych miejscach w pamięci, najlepiej oddalonych od siebie. Cała sztuka polega na częstych zmianach R. W regularnych odstępach czasu, na przykład co 100 ms, generujemy nową wartość R' i aktualizujemy pamięć tak, by zawierała $R \oplus R'$ oraz $R \oplus R' \oplus m$. Sposób ten daje gwarancję, że każdy bit pamięci będzie zależeć od ciągu bitów losowych. W celu skasowania pamięci zapiszemy jako nowe m ciąg zer, w wyniku czego oba obszary pamięci otrzymają te same, losowe dane.

Chcąc odczytać wartość m, musimy pobrać dane z obu lokalizacji i poddać je operacji XOR. Podczas zapisywania obliczamy XOR nowych danych z R i umieszczamy wynik pod drugim z używanych adresów.

Należy zwrócić uwagę na to, by ciągi bitów R i $R \oplus m$ nie zostały umieszczone w układzie RAM obok siebie. Bez informacji na temat działania RAM nie jest to oczywiste, ale w większości pamięci bity przechowuje się w prostokątnej macierzy, której przestrzeń adresowa składa się z części odpowiedzialnych za wiersz i kolumnę. Dwa fragmenty przechowywane w miejscach, których adresy różnią się o `0x5555`, mają małą szansę, by sąsiadowały w układzie fizycznie (zakładamy przy tym, że w pamięci bity z adresami parzystymi nie są numerami wierszy, a z bity adresami nieparzystymi nie są numerami kolumn; zresztą takiej konstrukcji pamięci nigdy jeszcze nie widzieliśmy). Jeszcze lepsze rozwiązanie polegałoby na ustaleniu dwóch losowych adresów w bardzo dużej przestrzeni adresowej. W ten sposób prawdopodobieństwo przylegania dwóch wybranych obszarów byłoby bardzo małe, niezależnie od konstrukcji użytej pamięci.

Opisane rozwiązanie jest jedynie częściowe, przy tym jest ono jest dosyć trudne do zastosowania. Sprawdza się ono tylko dla niewielkich porcji danych, gdyż w przypadku większych ilości danych funkcja aktualizująca wartość działałaby zbyt wolno. Jednak jego użycie gwarantuje, że żaden obszar pamięci nie będzie narażony na regularne przechowywanie w nim tajnych danych w sposób wpływający na jego preferowany stan.

Nadal jednak nie mamy gwarancji, że pamięć zostanie skasowana. Z dokumentacji układów scalonych RAM nie wynika, że dane raz umieszczone w pamięci w ogóle muszą być kasowane. Oczywiście żaden układ nie zachowują się w ten sposób, ale z uważnej lektury wynika, że możliwe jest uzyskanie jedynie pewnego poziomu bezpieczeństwa.

Skoncentrowaliśmy się tutaj na pamięci głównej. To samo rozwiązanie zadziała w pamięci podręcznej, tyle że niemożliwe będzie kontrolowanie adresu, pod którym dane zostaną umieszczone. Analogiczne rozwiązanie nie działa w przypadku rejestrów procesora, ale rejestry używane są na tyle intensywnie, że proces przetrzymywania danych prawdopodobnie nigdy nie będzie ich dotyczył. Z drugiej strony rejestry rozszerzające, na przykład rejestry zmiennoprzecinkowe czy rejestry typu MMX, używane są rzadziej, więc mogą sprawiać pewne problemy.

W sytuacji, kiedy konieczne jest przechowanie dużej ilości danych, rozwiązanie polegające na utrzymywaniu dwóch kopii i mnożeniu ich przez nowe łańcuchy losowe staje się zbyt kosztowne. Lepsze okazuje się wtedy zaszyfrowanie dużego bloku danych i zapisanie w pamięci tekstu

[3] Nazwa pochodzi z poematu Lewisa Carrolla *The Hunting of the Snark* [15]. W polskiej terminologii: *Bądziol*; przekład Stanisława Barańczaka pt. *Łowy na Snarka — przyp. tłum.*

zaszyfrowanego, który ewentualnie mógłby zostać przez kogoś odczytany. Należy jedynie unikać przetrzymywania w pamięci kluczy, posługując się np. techniką Boojum. Więcej szczegółów czytelnik znajdzie w [24].

9.3.5. Dostęp osób postronnych

Przechowywanie poufnych danych ma jeszcze jeden aspekt: dotyczy on innych programów, które działając na tym samym komputerze mogą sięgać do naszych danych. Niektóre systemy operacyjne pozwalają kilku programom współużytkować pamięć. Istnienie innego programu, będącego w stanie odczytać tajne klucze, oznacza poważne zagrożenie. Często stosowane rozwiązanie, polegające na konieczności zainicjowania współdzielonej pamięci przez oba programy, znacznie ogranicza ryzyko. Inne przypadki obejmują możliwość automatycznego współużytkowania pamięci w wyniku współużytkowania biblioteki.

Szczególnie niebezpieczną klasą programów są środowiska kontrolowanego wykonywania programów (*debugery*). Używane obecnie systemy operacyjne często zawierają opcje przeznaczone właśnie dla debugerów. Różne wersje Windows pozwalają debugerom przejmować kontrolę nad już działającym procesem. Debugery mają bardzo szerokie możliwości, mogą między innymi odczytywać pamięć. System Unix umożliwia wykonanie zrzutu pamięci programu (ang. *core dump*). Zrzut pamięci jest to plik zawierający obraz pamięci z danymi programu, oczywiście włącznie z wszystkimi danymi poufnymi.

Inne niebezpieczeństwo zagraża ze strony użytkowników o szczególnie szerokich uprawnieniach. Nazywa się ich *superużytkownikami* albo *administratorami*. Wolno im robić rzeczy niedostępne dla zwykłych użytkowników. W systemie Unix na przykład administrator ma prawo odczytać dowolny fragment pamięci.

Ogólnie rzecz biorąc, program nie może skutecznie bronić się przed opisanymi typami ataków. Staranny projekt przyczyni się do wyeliminowania niektórych ze wspomnianych problemów, ale często okazuje się, że mamy raczej ograniczone możliwości. Tym niemniej wszystkie wymienione zagrożenia należy rozpatrzyć w kontekście konkretnego środowiska operacyjnego, w którym będzie pracować nasz projekt.

9.3.6. Integralność danych

Oprócz konieczności utrzymania tajemnic musimy jeszcze pamiętać o ochronie spójności przechowywanych danych. Funkcja MAC zabezpiecza spójność danych podczas transmisji, ale nie rozwiązuje problemu modyfikacji danych bezpośrednio w pamięci.

W naszym wywodzie zakładamy, że korzystamy z niezawodnego sprzętu. Gdyby było inaczej, niewiele dałoby się zrobić w kwestii bezpieczeństwa. Wobec tego warto poświęcić czas i środki na analizę niezawodności, choć w zasadzie należy ona do zadań systemu operacyjnego. Na pewno warto sprawdzić, czy nasza maszyna jest wyposażona w pamięć główną typu ECC (tzn. z obsługą korekcji błędów)[4]. W razie wystąpienia przekłamania w jednym bicie, mechanizm ECC wykryje i skoryguje błąd. Bez ECC każdy błąd powoduje, że procesor otrzyma błędne dane.

[4] Trzeba się upewnić, że wszystkie części składowe komputera obsługują pamięć ECC. Ostrzegamy zwłaszcza przed nieznacznie tańszymi modułami pamięci, które nie przechowują dodatkowych informacji, tylko obliczają je na bieżąco. Ich użycie całkowicie przekreśla celowość korzystania z pamięci ECC.

Czemu jest to aż tak ważne? Współczesne komputery przetwarzają ogromne liczby bitów. Załóżmy, że z pamięć jest bardzo dobra jakościowo i szansa błędnego odczytu z pojedynczego bitu pamięci w ciągu sekundy wynosi jedynie 10^{-15}. W maszynie wyposażonej w 128 MB, to znaczy w około 10^{12} bitów pamięci, błędu można spodziewać się co 1000 sekund, to znaczy w przybliżeniu co 17 minut. Taka ilość błędów jest z naszego punktu widzenia niedopuszczalna. Liczba błędów zwiększa się wraz ze wzrostem ilości pamięci w komputerze, tak więc w maszynie dysponującej 1 GB pamięci sytuacja tylko się pogorszy. W serwerach zwykle montuje się pamięć ECC, gdyż serwery mają dużo pamięci i pracują dłużej niż stacje robocze. Dobrze byłoby jednak utrzymać podobny poziom stabilności we wszystkich używanych komputerach.

Nasze uwagi dotyczą oczywiście kwestii sprzętowych. Zazwyczaj nie mamy możliwości określania rodzaju pamięci w komputerze, na którym nasza aplikacja będzie uruchamiana.

Czasami zagrożenie poufności naszych danych zagraża jednocześnie ich integralności. Na przykład za pomocą debugera można zmodyfikować stan pamięci należącej do programu. Superużytkownik ma też możliwość bezpośredniej modyfikacji pamięci. Nie da się zapobiec takiemu niebezpieczeństwu ani mu zaradzić, ale trzeba zdawać sobie z niego sprawę.

9.3.7. Co robić

Zachowanie tajemnicy we współczesnym komputerze nie jest takie proste, jak mogłoby się wydawać. Jest wiele dróg, którymi tajemnice mogą wyciekać z systemy. Dla pełnego bezpieczeństwa należałoby zablokować je wszystkie. Niestety, stosowane obecnie systemy operacyjne i języki programowania nie dostarczają stosownych narzędzi. Można i trzeba robić to, co się da, lecz oznacza to mnóstwo pracy ściśle związanej z konkretnym środowiskiem realizacji.

Opisane problemy powodują, że bardzo trudno jest stworzyć bibliotekę funkcji kryptograficznych. Zachowanie tajemnic wymaga często modyfikacji głównego programu aplikacji. Oczywiście program główny też zawiera dane, które powinny pozostać ukryte; w przeciwnym wypadku biblioteka kryptograficzna byłaby w ogóle zbędna. Wracamy do znanego już stwierdzenia, że zachowanie bezpieczeństwa w systemie jako całości wymaga zachowania go w każdej części tego systemu.

9.4. Jakość kodu źródłowego

Podczas realizacji systemu kryptograficznego warto wiele czasu poświęcić jakości kodu programu. Mimo, że nasza książka ta nie dotyczy bezpośrednio programowania, powiemy kilka słów na ten temat, gdyż zagadnienia jakości kodu są zazwyczaj pomijane w książkach poświęconych programowaniu.

9.4.1. Prostota

Złożoność jest największym to wrogiem bezpieczeństwa. Wobec tego w projektach związanych z bezpieczeństwem należy za wszelką cenę dążyć do prostoty. Należy eliminować wszystkie elementy, które nie są absolutnie niezbędne. Trzeba pozbyć się wszystkich barokowych ozdobników, na ogół zresztą wykorzystywanych przez bardzo niewielu użytkowników. Należy z dala omijać

projekty zbiorowe, gdyż praca w komisji zawsze wymusza dodatkowe opcje w ramach kompromisu. Tam, gdzie w grę wchodzi bezpieczeństwo, prostota jest warunkiem pierwszorzędnym.

Typowym przykładem jest nasz bezpieczny kanał. Jego projekt nie zawiera żadnych opcji. Nie pozwala zaszyfrować nieuwierzytelnionych danych ani uwierzytelnić danych niezaszyfrowanych. Wielu użytkowników byłoby zadowolonych z dodatkowych możliwości, ale zwykle są to osoby nieświadome konsekwencji takich rozwiązań. Większość użytkowników wie zbyt mało o bezpieczeństwie, by pozwolić im wybierać odpowiednie opcje zabezpieczeń. Najlepszym rozwiązaniem jest rezygnacja z opcji i wymuszanie bezpieczeństwa domyślnie. Jeśli jest to naprawdę konieczne, należy umożliwić tylko jeden wybór: albo bezpieczny, albo niebezpieczny.

Wiele systemów udostępnia cały szereg szyfrów, zaś użytkownik może wybrać do użytku szyfr i funkcję uwierzytelniającą. Nawet w sytuacji, która teoretycznie dopuszcza taki wybór, zalecamy bezwzględne eliminowanie podobnych komplikacji. W zamian należy wybrać jeden tryb działania dostatecznie bezpieczny do wszystkich możliwych zastosowań. Różnice złożoności obliczeniowej poszczególnych trybów szyfrowania są niewielkie, a obecnie kryptografia już tylko wyjątkowo bywa wąskim gardłem przetwarzania. Pozbywając się opcji nie tylko pozbywamy się złożoności, ale jednocześnie uniemożliwiamy użytkownikowi takie skonfigurowanie aplikacji, by stosował w niej słabe szyfry. Ostatecznie skoro wybór trybu szyfrowania i autoryzacji jest tak trudny, że projektant nie podejmuje ostatecznej decyzji, to na jakiej podstawie miałby ją podjąć użytkownik?

9.4.2. Modularyzacja

Nawet po wyeliminowaniu mnóstwa opcji i dodatkowych możliwości może się okazać, że uzyskany system nadal pozostaje dość złożony. Zapanowanie nad złożonością umożliwia technika modularyzacji. Dzieli się w niej system na odrębne moduły, które następnie będą osobno projektowane, analizowane i implementowane.

Sama koncepcja modularyzacji z pewnością nie jest obca Czytelnikowi; w kryptografii po prostu nabiera wagi właściwe jej zastosowanie. Mówiąc wcześniej o elementach składowych systemu kryptograficznego, odnosiliśmy się do nich jak do modułów. Interfejs modułu powinien być prosty, powinien też zachowywać się zgodnie z intuicją. Warto przyjrzeć się interfejsom zaprojektowanych przez siebie modułów. Często można w nich dostrzec właściwości czy opcje, które służą do rozwiązania problemów związanych z innym modułem. Jeśli tylko to możliwe, należy je odrzucić. Każdy moduł powinien zajmować się wyłącznie sam sobą. Z naszego doświadczenia wynika, że kiedy moduł zaczyna obrastać w dziwne właściwości, nadchodzi czas na przejrzenie projektu, gdyż prawie zawsze okazuje się, że owe dziwne cechy były związane z jakimiś wadami projektowymi.

Modularyzacja jest tak istotna, ponieważ jest jedyną skuteczną metodą panowania nad złożonością. Dana opcja ograniczona do pojedynczego modułu może być przeanalizowana w kontekście tego modułu. Jednak opcja, która zmienia zewnętrzne zachowanie danego modułu, może wpływać na zachowanie innych modułów. W układzie 20 modułów, z których każdy jest wyposażony w jedną dwustanową opcję zmieniającą jego działanie, możliwych jest ponad milion konfiguracji. Dla pełnego bezpieczeństwa konieczne byłoby przeanalizowanie wszystkich tych możliwości, co jednak jest niemożliwe.

Jak zauważyliśmy, że wiele opcji powstaje z myślą o poprawie wydajności. Temat ten jest dobrze znany w inżynierii oprogramowania. Wiele systemów jest wyposażonych w tak zwane optymalizacje, które jednak okazują się całkiem bezużyteczne, zmniejszając wydajność, lub są nieistotne, gdyż optymalizują te części systemu, które nie są wcale wąskimi gardłami całości. W sprawach

optymalizacji jesteśmy bardzo zachowawczy. Zwykle w ogóle się nią nie przejmujemy. Tworzymy dokładny projekt, a następnie staramy się zapewnić, by dało się go zrealizować w dużych kawałkach. Za typowy przykład niech posłuży stary BIOS IBM PC. Procedura wyświetlająca znak na ekranie ma tylko jeden parametr — wyświetlany znak. Procedura spędza większość swojego czasu bezproduktywnie, a samo wstawiane znaku do bufora pamięci zajmuje tylko drobną jego część. Gdyby interfejs procedury pozwalał użyć w charakterze parametru łańcucha znaków, to możliwe byłoby wyświetlenie całego łańcucha w czasie nieznacznie dłuższym niż wyświetlenie pojedynczego znaku. Wynikiem tak kiepskiego projektu było bardzo powolne wyświetlanie znaków w systemie DOS. Ta sama zasada obowiązuje w projektach kryptograficznych. Należy dążyć do tego, by przetwarzanie odbywało się w możliwie dużych porcjach. Pozostanie wtedy jedynie optymalizacja tych części programu, których wpływ na wydajność całości zostanie *wykazany* jako istotny.

9.4.3. Asercje

Asercje są doskonałym narzędziem do poprawy jakości programów[5].

Podczas implementacji programu kryptograficznego trzeba przyjąć postawę zawodowego paranoika. Żaden moduł nie ma prawa ufać innym modułom, wszystkich obowiązuje sprawdzanie poprawności parametrów, wymuszanie ograniczeń wywoływania oraz odmowa wykonywania niebezpiecznych operacji. W większości sytuacji chodzi o bardzo proste asercje. Jeżeli specyfikacja modułu wymaga, by użycie pewnego obiektu było poprzedzone jego inicjalizacją, to odwołanie do niezainicjowanego obiektu tego typu powinno spowodować wygenerowanie błędu asercji. Błędy związane z niespełnieniem asercji powinny zawsze prowadzić do przerwania programu i udostępnienia diagnozy błędu.

Jako ogólną zasadę przyjmujemy, że każda kontrola spójności wewnątrz programu winna posługiwać się asercjami. Należy wychwytywać tym sposobem tyle błędów, ile tylko się da; dotyczy to zarówno błędów własnych, jak i innych programistów. Błąd wychwycony przez asercję oznacza uniknięcie dziury w systemie bezpieczeństwa.

Niektórzy programiści realizują asercje na etapie budowy oprogramowania, a wyłączają je przed oddaniem ostatecznej wersji. Kto to wymyślił? Co by było, gdyby pociągi do nuklearnej elektrowni podjeżdżały zgodnie z pełną procedurą ochronną, ale przy podjeżdżaniu do samego reaktora wyłączały systemy bezpieczeństwa. Albo co myśleć o spadochroniarzu, który na każdy trening naziemny zabierałby spadochron zapasowy, ale z samolotu skakałby już bez niego?. Po co w ogóle usuwać sprawdzanie asercji z programów przekazywanych klientowi? Przecież tylko tam jest ono naprawdę potrzebne! Kiedy asercja zawodzi w gotowym kodzie wykonywalnym, mamy do czynienia z błędem programistycznym. Zignorowanie tego błędu prawdopodobnie doprowadzi do dalszych błędów, gdyż przynajmniej jedno założenie dotyczące programu zostało naruszone. Podawanie nieprawidłowych odpowiedzi jest chyba najgorszym możliwym sposobem działania programu. Znacznie lepiej jest poinformować użytkownika, że wystąpił błąd programistyczny, wobec czego nie należy bezgranicznie ufać dalszym odpowiedziom systemu. Szczegółowe sprawdzanie błędów zostawimy już Czytelnikom.

[5] Wiemy, że nasze uwagi upodabniają książkę do niedzielnej szkółki programowania, ale przypominamy je wszystkim programistom, z którymi pracujemy.

9.4.4. Przepełnienie bufora

Sam fakt, że jesteśmy zmuszeni umieścić w książce bieżący podrozdział, przynosi wstyd całemu przemysłowi informatycznemu. Problemy związane z przepełnieniem bufora znane są już od 40 lat. Od równie długiego czasu znane są skuteczne sposoby ich unikania. Niektóre wczesne języki programowania wysokiego poziomu, na przykład Algol 60, całkowicie eliminowały ten problem wprowadzając obowiązkową kontrolę zakresu indeksów tablic. Mimo to przepełnienie bufora jest przyczyną około połowy przypadków naruszenia bezpieczeństwa w Internecie. Błędy tego rodzaju wciąż się zdarzają, gdyż nikt nie ma ochoty używać lepszych narzędzi. Uważamy, że jest to przejaw karygodnego lekceważenia obowiązków, porównywalnego do postawy producenta samochodów, który robiłby zbiorniki na paliwo z woskowanego papieru. Póki wszystko działa poprawnie, nikt oczywiście nie będzie się czepiał, ale naszym zdaniem dyrektor odpowiedzialny za taką praktykę powinien trafić do więzienia. Z niejasnych powodów duża część firm informatycznych zachowuje się tak, jakby nie ponosiła odpowiedzialności za konsekwencje swoich działań (być może wynika to stąd, że prawodawcy umożliwili unikanie odpowiedzialności za pomocą takiej konstrukcji umów licencyjnych, która byłaby nie do pomyślenia w jakiejkolwiek innej dziedzinie przemysłu). Biorąc pod uwagę, że takie lekceważenie jest udziałem znakomitej większości uczestników rynku informatycznego, zastanawiamy się czasem, czy dziedzina tak zaawansowana, jak kryptografia, jest w ogóle warta podejmowania wysiłku.

Jednak nie jesteśmy w stanie zmienić wszystkiego. Możemy jedynie doradzić, jak pisać dobre programy kryptograficzne. Należy unikać języków programowania pozwalających na przepełnianie buforów. Dokładniej mówiąc, nie należy używać C ani C++. Nigdy nie należy wyłączać kontroli indeksów tablic w kompilatorze języka programowania. Zasada jest naprawdę prosta, lecz jej nieprzestrzeganie jest prawdopodobnie przyczyną około połowy błędów związanych z bezpieczeństwem.

9.4.5. Testowanie

Dokładne testowanie jest obowiązkowym elementem procesu tworzenia oprogramowania. Testowanie pomaga w znalezieniu błędów w programach, ale nie pozwoli ono znaleźć dziur w systemie zabezpieczeń. Nigdy nie należy mylić testowania z analizą bezpieczeństwa. Te dwa elementy wzajemnie się uzupełniają, ale są odrębne.

Istnieją dwa rodzaje testów, które należałoby realizować. Pierwszy obejmuje ogólne testy związane ze specyfikacją funkcjonalną modułu. W idealnej sytuacji jeden programista implementuje moduł, a drugi realizuje testy. Obaj pracują na podstawie specyfikacji funkcjonalnej. Każde nieporozumienie między nimi wskazuje na konieczność poprawienia specyfikacji. Testy ogólne powinny obejmować wszystkie możliwości modułu. Czasami testy są proste; a czasami program testujący musi zastąpić całe środowisko. Kod programu testującego często bywa równie obszerny, jak sam kod testowanego modułu, i nie ma na to rady.

Druga grupa testów obejmuje testy wykonywane przez samego programistę realizującego dany moduł. Testy te służą do zbadania wszelkich ograniczeń danej implementacji. Na przykład dodatkowe testy modułu, który używa wewnętrznego bufora o rozmiarze 4 kB, powinny obejmować traktowanie skrajnych części bufora, tak by na ich podstawie dało się wychwycić wszelkie błędy zarządzania tym buforem. Czasem stworzenie odpowiednich testów wymaga znajomości wewnętrznej budowy modułu.

Często tworzymy ciągi testów sterowane generatorem wartości losowych. Generator liczb losowych omówimy w rozdziale 10. Użycie takiego generatora znakomicie ułatwia przeprowadzenie wielu testów. Przechowanie wartości ziarna generatora pozwala powtórzyć cały cykl testów, co jest przydatne przy wykrywaniu i usuwaniu błędów. Szczegóły realizacji zależą od konkretnego modułu.

W końcu dobrze jest mieć program do „szybkiego testowania", który można by uruchamiać przy każdym uruchamianiu programu. W swoich ostatnich projektach jeden z autorów, Niels, miał zaimplementować AES. Kod inicjalizujący wykonywał algorytm AES dla kilku przypadków testowych i sprawdzał, czy wyniki są zgodne ze znanymi prawidłowymi odpowiedziami. Każde naruszenie poprawności kodu implementującego AES, które może się zdarzyć w toku dalszego rozwoju aplikacji, zostanie prawdopodobnie wychwycone przez opisany przed chwilą szybki test.

9.5. Ataki bocznym kanałem

Znana jest cała klasa ataków, noszących nazwę ataków bocznym kanałem [49]. Ataki te są możliwe, kiedy atakujący dysponuje dodatkowym kanałem informacji o systemie. Np. atakujący może dokładnie mierzyć czas, jaki zajmuje zaszyfrowanie wiadomości. Sprzętowa implementacja składnika kryptograficznego w postaci karty procesorowej pozwala zmierzyć, ile energii ta karta zużywa. Pola magnetyczne, emisja promieniowania podczerwonego, zużycie prądu, czas odpowiedzi, interferencja z innymi kanałami danych — wszystkie te dane mogą posłużyć do przeprowadzenia ataku bocznym kanałem.

Nie powinno zaskakiwać, że tego typu ataki okazują się najskuteczniejsze w odniesieniu do systemów, przy których tworzeniu nie uwzględniono tej klasy ataków. Szczególnie skuteczna jest analiza zużycia prądu przez karty procesorowe [57].

Ochrona przed wszystkimi formami ataków bocznym kanałem jest bardzo trudna, o ile w ogóle możliwa, tym niemniej zawsze da się przedsięwziąć pewne proste środki ostrożności. Przed laty Niels implementował system zabezpieczeń kart procesorowych, w którym jedna z zasad projektowych głosiła, że ciąg instrukcji wykonywanych przez procesor zależeć może jedynie od informacji znanych już atakującemu. Ten warunek wyklucza ataki oparte na analizie czasu odpowiedzi, utrudnia też analizę zużycia prądu, gdyż wykonywany ciąg instrukcji nie ujawnia już żadnych nowych informacji. Warunek ów nie opisuje pełnego rozwiązania problemu, poza tym współczesne techniki analizy zużycia mocy potrafią bez problemu złamać zabezpieczenia kart tworzonych w czasach, o których jest tu mowa. Tym niemniej rozwiązanie, jakie wtedy zaproponowaliśmy, było najlepszym z realnie możliwych. Odpieranie ataków bocznym kanałem zawsze polega na pewnej kombinacji przeciwdziałań — z których część jest zaimplementowana w oprogramowaniu systemu kryptograficznego, a część ma realizację sprzętową.

Ochrona przed atakami bocznym kanałem ma wszelkie cechy wyścigu szczurów. Staramy się chronić przed znanymi kanałami bocznymi, lecz ktoś dostatecznie sprytny znajduje nowy kanał. Wtedy jesteśmy zmuszeni się cofnąć i uwzględnić także nowy kanał. W praktyce nie jest aż tak źle, gdyż większość ataków bocznym kanałem jest bardzo trudna do przeprowadzenia. Kanały boczne są realnym zagrożeniem dla kart procesorowych, gdyż przeciwnik po przechwyceniu takiej karty może poddać ją pełnej analizie. Jednak w przypadku innych rodzajów komputerów praktyczne znaczenie mają tylko niektóre rodzaje kanałów. Z praktycznego punktu widzenia najważniejszymi kanałami bocznymi są: czas reakcji oraz nasłuch radiowy (karty procesorowe są szczególnie podatne na pomiar zużycia prądu).

9.6. Wnioski

Mamy nadzieję, że lektura bieżącego rozdziału wyjaśniła, że bezpieczeństwo nie zaczyna się ani nie kończy na projekcie kryptograficznym. Na bezpieczeństwo wpływają istotnie wszystkie aspekty funkcjonowania całego systemu. Dlatego specjaliści bezpieczeństwo spotykają się tak często z negatywnymi reakcjami: wszędzie wściubiają swoje nosy, wszystkich pouczają, a w końcu odcinają mnóstwo bardzo przydatnych opcji, twierdząc, jakoby były one niebezpieczne.

Implementacja systemu kryptograficznego sama w sobie jest sztuką. Najważniejsza jest jakość kodu programu. Kod niskiej jakości jest w praktyce najczęstszym powodem ataków; z drugiej strony wyeliminowanie kiepskiego kodu jest stosunkowo łatwe. Z naszego doświadczenia wynika, że tworzenie porządnego kodu źródłowego wcale nie trwa dłużej niż tworzenie kodu byle jakiego, o ile tylko liczy się czas od początku projektu do oddania gotowego produktu, a nie do czasu oddania pierwszej, najeżonej błędami, wersji. Warto fanatycznie dbać o jakość tworzonego kodu. Porządne kodowanie leży jak najbardziej w zasięgu możliwości i trzeba się go nauczyć, możliwie jak najprędzej.

Idealnie byłoby przeanalizować i przebudować całe środowisko, w tym stosowany język programowania i system operacyjny, jako najwyższy priorytet przyjmując bezpieczeństwo. Mamy wielką ochotę wziąć kiedyś udział w takim projekcie, więc prosimy o kontakt każdego, ktoś ma do wydania kilka milionów dolarów na komputer *naprawdę* godny zaufania.