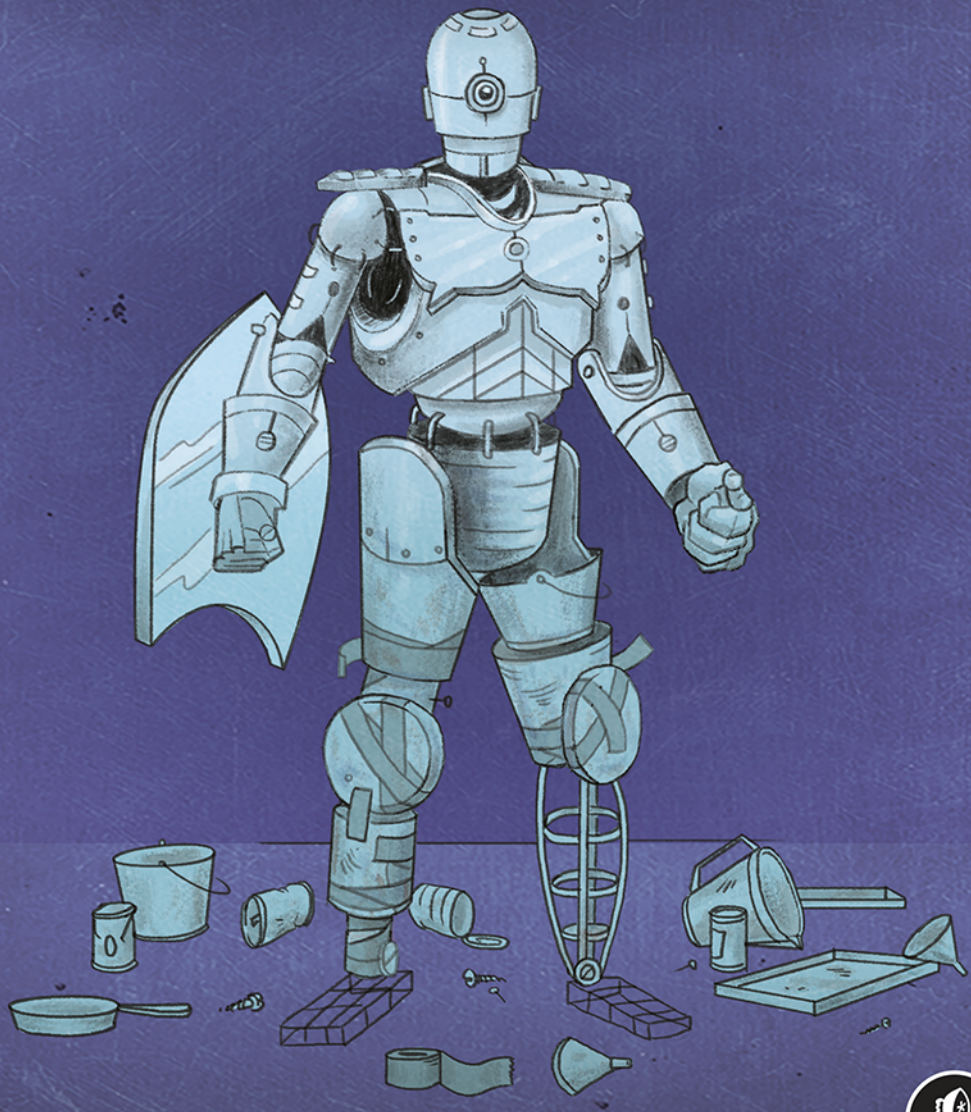
Po pierwsze: bezpieczeństwo

Przewodnik dla twórców oprogramowania



Helion

Loren Kohnfelder

no starch press

Tytuł oryginału: Designing Secure Software A Guide for Developers

Tłumaczenie: Magdalena Tkacz

ISBN: 978-83-283-9432-2

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

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

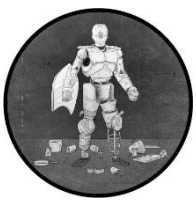
Spis treści

4

Wzorce

Wzorzec tworzony z uczuciem staje się dziełem sztuki.

— Herbert Read



Architekci od dawna stosują wzorce projektowe do projektowania nowych budynków. Takie podejście jest również przydatne w projektowaniu oprogramowania. W rozdziale przedstawiono wiele najbardziej przydatnych wzorców wspomagających tworzenie bezpiecznych projektów. Wiele z nich wywodzi się ze starożytnych mądrości. Sztuką jest wiedzieć, jak je zastosować w tworzeniu oprogramowania i w jaki sposób ich wykorzystanie wpływa na zwiększenie jego bezpieczeństwa.

Wzorce te wpływają na złagodzenie lub uniknięcie różnych luk w zabezpieczeniach, są ważnym zestawem narzędzi do radzenia sobie z potencjalnymi zagrożeniami. Niektóre z nich są proste, ale inne mogą być trudniejsze do zrozumienia i lepiej wyjaśnić je na przykładach. Nie należy jednak lekceważyć tych prostszych, ponieważ można je szerzej wykorzystać i są równocześnie jednymi z najbardziej efektywnych. Inne koncepcje mogą być łatwiejsze do zrozumienia jako antywzorce, w których pokazano, czego *nie należy robić*. Przedstawiam te wzorce pogrupowane według wspólnych cech, które można traktować jako sekcje zestawu narzędzi (rysunek 4.1).

Rysunek 4.1. Grupy wzorców występujących w bezpiecznym oprogramowaniu omawiane w tym rozdziale

To, kiedy i gdzie zastosować który wzorzec, wymaga zdroworozsądkowego podejścia. Decyzje projektowe najlepiej podjąć w oparciu o bieżące potrzeby projektu, przy czym te najprostsze są najlepsze. Podobnie jak w świecie fizycznym nie potrzebujesz siedmiu zamków i łańcuchów przy drzwiach, tak nie musisz stosować wszystkich możliwych wzorców projektowych, aby rozwiązać problem. Jeśli można zastosować kilka wzorców, należy wybrać jeden lub dwa najlepsze, a w przypadku krytycznych wymagań dotyczących bezpieczeństwa — więcej. Nadużywanie wzorców może przynieść efekty odwrotne do zamierzonych, ponieważ w dziedzinie bezpieczeństwa malejące zyski (wynikające z większej złożoności i kosztów ogólnych) szybko przewyższają dodatkowe korzyści.

Cechy projektu

Pierwsza grupa wzorców na wysokim poziomie abstrakcji opisuje, jak wygląda bezpieczny projekt; ma być prosty i przejrzysty. Wywodzi się to ze znanych powiedzeń „nie komplikuj" (ang. *keep it simple*) oraz „nie powinieneś mieć nic do ukrycia" (ang. *you should have nothing to hide*). Reguły te jako podstawowe i oczywiste mogą być stosowane na szeroką skalę. Są również bardzo skuteczne.

Ekonomia projektowania

Projekt powinien być jak najprostszy.

Ekonomia w procesie tworzenia projektu podnosi poprzeczkę bezpieczeństwa, ponieważ w prostszych projektach mniej prawdopodobne jest występowanie wielu błędów, a tym samym będzie mniej nieodkrytych podatności. Chociaż programiści twierdzą, że „każde oprogramowanie ma błędy", wiemy, iż proste programy z pewnością mogą być wolne od błędów. Podczas projektowania mechanizmów bezpieczeństwa należy preferować jak najprostsze konstrukcje i wystrzegać się skomplikowanych w przypadku pełnienia krytycznych funkcji związanych z bezpieczeństwem.

Doskonałym przykładem takiego wzorca są klocki LEGO. Gdy projekt i produkcja standardowego elementu konstrukcyjnego zostaną dopracowane do perfekcji, umożliwia to budowanie niezliczonej liczby kreatywnych konstrukcji. Podobnie system składający się z wielu mało uniwersalnych elementów jest trudniejszy do zbudowania. Każdy nowy projekt wymaga wtedy większej liczby komponentów i wiąże się z dodatkowymi wyzwaniami technicznymi.

Wiele przykładów ekonomii projektowania można znaleźć w architekturze systemów dużych usług internetowych, które są tworzone z myślą o działaniu w ogromnych centrach danych. Aby zapewnić ich niezawodne działanie na taką skalę, poszczególne funkcje są dzielone na mniejsze, samodzielne komponenty, które wspólnie wykonują skomplikowane operacje. Często bazowy frontend obsługuje zakończenie żądania HTTPS, przetwarzając i walidując przychodzące dane przed przekazaniem ich do wewnętrznej struktury. Ta struktura danych jest następnie przesyłana dalej do dalszego przetwarzania przez szereg usług podrzędnych, które z kolei wykorzystują mikrousługi do wykonywania różnych funkcji.

W aplikacjach, takich jak wyszukiwarka internetowa, różne maszyny mogą niezależnie budować równolegle różne części odpowiedzi, po czym jeszcze inne łączą je w kompletną odpowiedź. Dużo łatwiej jest zbudować wiele małych usług wykonujących poszczególne części całego zadania — takich jak: parsowanie zapytań, korekta pisowni, wyszukiwanie tekstu, wyszukiwanie obrazów, ranking wyników i układ strony — niż zrobić to wszystko w jednym ogromnym programie.

Ekonomia projektowania nie nakazuje, że wszystko musi być zawsze proste. Podkreśla raczej ogromne zalety prostoty i mówi, że złożoność należy stosować tylko wtedy, gdy wnosi ona znaczącą wartość dodaną. Rozważmy różnice między konstrukcją list kontroli dostępu (ang. *access control list*, ACL) w systemach *nix i Windows. Pierwsza z nich jest prosta i określa uprawnienia do odczytu/zapisu/wykonania dla użytkownika, grupy użytkowników lub dla wszystkich. Druga jest znacznie bardziej złożona, bo zawiera dowolną liczbę wpisów zezwalających na dostęp i odmawiających dostępu, a także funkcje dziedziczenia. Na dodatek rezultat zależy od kolejności wpisów na liście (te uproszczone opisy mają na celu zwrócenie uwagi na projekt i celowo nie są kompletne). Ten przykład pokazuje, że prostsze uprawnienia w systemie *nix są łatwiejsze do prawidłowego egzekwowania, a poza tym użytkownikom systemu łatwiej zrozumieć, jak działają, a co za tym idzie — jak ich prawidłowo używać. Jeśli jednak lista ACL systemu

Windows zapewnia właściwą ochronę danej aplikacji i można ją precyzyjnie skonfigurować — to też może być dobrym rozwiązaniem.

Wzorzec Ekonomii projektowania nie mówi jednoznacznie, że prostsza opcja jest zawsze lepsza ani że bardziej złożona stanie się bez wątpienia problematyczna. W przytoczonym przykładzie widać, że listy ACL systemu *nix nie są z natury lepsze, a listy ACL systemu Windows nie muszą być pełne błędów. Jednak listy ACL systemu Windows wymagają od programistów i użytkowników więcej nakładu czasu i pracy, by się ich nauczyć, a stosowanie ich bardziej skomplikowanych funkcji może łatwo wprowadzić użytkowników w błąd i spowodować różne niezamierzone konsekwencje. Kluczowym wyborem, którego trzeba dokonać w trakcie tworzenia projektu (którego nie będę tu rozstrzygał), jest to, które listy i w jakim stopniu listy ACL najlepiej odpowiadają potrzebom użytkowników. Być może listy ACL systemu *nix są zbyt proste i nie odpowiadają rzeczywistym wymaganiom; z drugiej zaś strony, może się okazać, że listy ACL systemu Windows są zbyt rozbudowane i uciążliwe w typowych zastosowaniach. Są to trudne pytania, na które każdy z nas musi sobie na własny użytek odpowiedzieć i na które ten wzorzec projektowy pozwala spojrzeć z różnych stron.

Przejrzysty projekt

Silna ochrona nigdy nie powinna opierać się na tajemnicy.

Prawdopodobnie najbardziej znanym przykładem projektu, w którym nie stosowano zasady **Przejrzystego projektu** (ang. *transparent design*), jest Gwiazda Śmierci z filmu *Gwiezdne wojny*, której szyb termowentylacyjny umożliwiał uderzenie prosto w serce stacji bojowej. Gdyby Darth Vader rozliczał swoich architektów z przestrzegania tej zasady równie surowo jak admirała Mottiego, historia potoczyłaby się zupełnie inaczej. Jawność projektu dobrze zbudowanego systemu powinna zniechęcić napastników, pokazując jego niezwyciężoność, a nie ułatwiać im zadania. Odpowiadający mu antywzorzec jest być może lepiej znany: nazywamy go **Bezpieczeństwem przez zaciemnianie** (ang. *security by obscurity*).

We wzorcu tym mamy wyraźne ostrzeżenie, by *nie polegać na tym*, że szczegóły projektu nie są znane. Nie oznacza to, że publiczne ujawnianie projektu jest obowiązkowe ani że jest coś złego w tajnych informacjach. Jeśli pełna jawność projektu osłabi bezpieczeństwo, należy go naprawić, a nie liczyć na utrzymanie w tajemnicy. Nie dotyczy to w żadnym wypadku chronionych przepisami informacji, takich jak klucze kryptograficzne czy tożsamość użytkowników, których wyciek mógłby w istocie zagrozić bezpieczeństwu. Dlatego wzorzec ten ma nazwę „Przejrzysty projekt", a nie „Całkowicie transparentny". Pełne ujawnienie sposobu konstrukcji metody szyfrowania, np. wielkości klucza, formatu wiadomości, algorytmów kryptograficznych itd., nie powinno w ogóle osłabiać bezpieczeństwa. Ten antywzorzec powinien być ogromnym plakatem o przykładowej treści: „Nie ufaj wszystkim samozwańczym »ekspertom«, którzy twierdzą, że wynaleźli niesamowite algorytmy szyfrowania i są one tak wspaniałe, że nie mogą opublikować ich szczegółów". W każdym przypadku, bez wyjątku są to fałszywe deklaracje.

Problem z zapewnianiem bezpieczeństwa przez zaciemnienie polega na tym, że chociaż może ono na krótko pomóc w powstrzymaniu przeciwników, to jest niezwykle kruche. Wyobraźmy sobie np., że w projekcie użyto przestarzałego algorytmu kryptograficznego. Jeśli atakujący dowiedzieliby się, że oprogramowanie nadal używa powiedzmy DES (starszy symetryczny algorytm szyfrowania z lat 70. ubiegłego wieku), mogliby z łatwością złamać go w ciągu jednego dnia. Zamiast rekomendować w projekcie stary algorytm, należy wykonać niezbędne prace pozwalające uzyskać solidne podstawy bezpieczeństwa, tak aby nie było nic do ukrycia (niezależnie od tego, czy szczegóły projektu są publiczne, czy nie).

Minimalizacja narażenia

Największa grupa wzorców nakazuje ostrożność: należy „dmuchać na zimne". Jest to przejaw podstawowej strategii ryzyko/zysk, którą możesz bezpiecznie stosować — chyba że istnieje ważny powód, aby postąpić inaczej.

Najmniejsze przywileje

Zawsze najbezpieczniej jest użyć tylko tylu przywilejów, ilu potrzeba do wykonania zadania.

Nigdy nie czyść załadowanej broni. Odłączaj pilarki elektryczne od zasilania podczas wymiany brzeszczotów. Te powszechnie stosowane praktyki bezpieczeństwa są przykładem wzorca **Najmniejszych przywilejów** (ang. *Least Privilege*), którego celem jest zmniejszenie ryzyka popełnienia błędu podczas wykonywania zadania. Ten wzorzec jest przyczyną, z powodu której administratorzy ważnych systemów nie powinni przeglądać przypadkowych stron w Internecie, wykonując swoją pracę[1]. Jeśli odwiedzą złośliwą witrynę i nastąpi infiltracja, łatwo o wyrządzenie poważnych szkód.

W systemach *nix właśnie do tego celu służy polecenie `sudo(1)`. Posługujący się kontami użytkowników z wysokimi przywilejami (tzw. *sudoers*) muszą być ostrożni, aby przypadkowo lub w razie infiltracji nie wykorzystać swoich nadzwyczajnych uprawnień. Aby zapewnić taką ochronę, użytkownik musi poprzedzić polecenia superużytkownika poleceniem `sudo`, które poprosi użytkownika o podanie hasła do konta, aby wykonać polecenie. W tych systemach większość poleceń (tych które nie wymagają `sudo`) ma wpływ tylko na swoje własne konto użytkownika i nie może mieć wpływu na cały system. Można to porównać do szybki na wyłączniku alarmu przeciwpożarowego „W PRZYPADKU AWARII ZBIJ SZYBKĘ". W ten sposób zapobiegamy przypadkowemu uruchomieniu, ponieważ wymuszane jest wykonanie wyraźnego kroku (odpowiadającego poleceniu `sudo`) przed uruchomieniem włącznika. Szybka sprawia, że nikt nie może twierdzić, że przypadkowo włączył alarm przeciwpożarowy. Podobnie jest z kompetentnym administratorem, który nigdy nie wpisze przez przypadek `sudo` i polecenia, które może zniszczyć system.

[1] Przy równoczesnym zalogowaniu do obsługiwanego systemu — *przyp. tłum.*

Ten wzorzec jest ważny z tego prostego powodu: kiedy luki są wykorzystywane, lepiej jest, aby atakujący miał minimalne uprawnienia, które mógłby wykorzystać w punkcie startu. Z niemal nieograniczonych uprawnień, takich jak uprawnienia superużytkownika, należy korzystać tylko wtedy, gdy jest to absolutnie konieczne i to przez jak najkrótszy czas. Nawet Superman stosował zasadę „najmniejszych uprawnień", zakładając swój „służbowy" strój tylko wtedy, gdy miał do wykonania zadanie, a po uratowaniu świata natychmiast stawał się z powrotem Clarkiem Kentem.

W praktyce selektywne i oszczędne korzystanie z podwyższonych uprawnień wymaga więcej wysiłku. Tak jak odłączenie od zasilania narzędzi elektrycznych w celu ich obsługi wymaga więcej wysiłku, tak samo ostrożność w używaniu uprawnień potrzebuje dyscypliny, ale robienie tego w dobrze przemyślany sposób jest zawsze bezpieczniejsze. W przypadku exploitu może to oznaczać różnicę pomiędzy niewielkim wtargnięciem do systemu a jego całkowitym zniszczeniem. Praktykowanie zasady najmniejszego przywileju może również zmniejszyć szkody wyrządzone przez ludzkie błędy i pomyłki.

Podobnie jak w przypadku wszystkich zasad, należy stosować ten wzorzec z wyczuciem, aby uniknąć nadmiernego komplikowania sytuacji. Reguła najmniejszych przywilejów nie oznacza, że system powinien zawsze przyznawać najmniejszy dostępny poziom uprawnień (np. przy tworzeniu kodu, który — aby zapisać plik X — otrzymuje dostęp umożliwiający zapis tylko tego jedynego pliku). Można się zastanawiać, dlaczego nie wykorzystywać tego doskonałego wzorca zawsze „na maksimum"? Oprócz zachowania ogólnego poczucia równowagi i dostrzegania malejących korzyści z każdego złagodzenia, ważnym czynnikiem jest tutaj szczegółowość mechanizmu kontrolującego autoryzację oraz koszty ponoszone podczas „dostrajania" uprawnień w górę i w dół. Przykładowo w procesach w systemach *nix uprawnienia są nadawane na podstawie list kontroli dostępu opartych na identyfikatorach użytkowników i grup. Poza elastycznym przełączeniem się między efektywnymi i rzeczywistymi identyfikatorami (właśnie to umożliwia `sudo`) nie mamy prostego sposobu na tymczasowe usunięcie niepotrzebnych uprawnień bez tworzenia kopii procesu. Kod powinien działać z niższymi przywilejami tam, gdzie jest to możliwe, a używać wyższych przywilejów w niezbędnych momentach i przechodzić do nich w oczywistych punktach decyzyjnych.

Jak najmniej informacji

Zawsze najbezpieczniej jest gromadzić jak najmniejszą ilość prywatnych informacji potrzebnych do wykonania zadania i uzyskiwać dostęp do tej jak najmniejszej ilości.

Wzorzec **Dostępu do jak najmniejszej ilości informacji** (ang. *least information*) jest odpowiednikiem wzorca Najmniejszych możliwych przywilejów (ang. *least privilege*) i pomaga zminimalizować ryzyko nieplanowanego ujawnienia informacji. Podczas wywoływania podprogramu żądania usługi lub odpowiedzi na żądanie należy unikać przekazywania większej ilości prywatnych informacji, niż jest to absolutnie konieczne, a także przy każdej okazji ograniczać niepotrzebny

przepływ informacji. Wdrożenie tego wzorca może być trudne w praktyce, ponieważ oprogramowanie zazwyczaj przekazuje standardowo „opakowane" dane. Te dodatkowe dołączane dane nie są optymalizowane pod kątem bezpieczeństwa i naprawdę nie są potrzebne.

Oprogramowanie zbyt często nie spełnia tego wzorca, ponieważ projekty interfejsów ewoluują z czasem, służąc wielu celom, a dla zachowania spójności wygodnie jest ponownie wykorzystywać te same parametry lub struktury danych. W rezultacie dane, które nie są absolutnie niezbędne, są przesyłane jako nadbagaż, który wydaje się nieszkodliwy. Problem pojawia się oczywiście wtedy, gdy te niepotrzebne dane przepływające przez system stwarzają dodatkowe możliwości ataku.

Wyobraźmy sobie np. duży system zarządzania relacjami z klientami (CRM) używany przez różnych pracowników przedsiębiorstwa. Różni pracownicy korzystają z systemu w wielu różnych celach, takich jak sprzedaż, produkcja, wysyłka, wsparcie, konserwacja, badania i rozwój oraz księgowość. W zależności od pełnionych ról każdy z nich ma inne uprawnienia dostępu do podzbiorów zawartych w systemie informacji. Aby zastosować zasadę Dostępu do jak najmniejszej ilości informacji, aplikacje w tym przedsiębiorstwie powinny żądać tylko minimalnej ilości danych potrzebnych do wykonania określonego zadania. Weźmy np. przedstawiciela działu obsługi klienta odpowiadającego na telefon. System używa identyfikatora rozmówcy do wyszukania rekordu klienta, a pracownik działu obsługi nie musi znać jego numeru telefonu, tylko historię zakupów. Porównaj to z prostszym projektem, który pozwala (lub nie) na wyszukiwanie rekordów klientów zawierających wszystkie pola danych. W idealnej sytuacji nawet jeśli pracownik ma szerszy dostęp do danych, powinien mieć możliwość zażądania minimum niezbędnego do wykonania swojego zadania i pracować w oparciu o to minimum, minimalizując w ten sposób ryzyko ujawnienia informacji.

Na poziomie implementacji projektowanie korzystające z wzorca Dostępu do jak najmniejszej ilości informacji obejmuje również usuwanie lokalnie zbuforowanych informacji, gdy nie są już potrzebne, lub wyświetlanie na ekranie tylko podzbioru dostępnych danych do momentu, gdy użytkownik wyraźnie zażąda zobaczenia pewnych szczegółów. Powszechna praktyka wyświetlania haseł w postaci ******** wykorzystuje ten wzorzec, aby zmniejszyć ryzyko ataku typu „zaglądanie przez ramię".

Szczególnie ważne jest, by zdecydować się na zastosowanie tego wzorca już na etapie projektowania, ponieważ później może być bardzo trudno go zaimplementować (gdyż zmianie muszą ulec obie strony interfejsu). Jeśli zaprojektujesz niezależne komponenty dostosowane do konkretnych zadań, które wymagają różnych zestawów danych, masz większe szanse, że uda Ci się to osiągnąć. Interfejsy API obsługujące dane wrażliwe powinny być na tyle elastyczne, aby umożliwić wywołaniom określenie podzbiorów (minimalizacja narażenia informacji) danych, których potrzebują (tabela 4.1).

Interfejs API `RequestCustomerData` przedstawiony w lewej kolumnie ignoruje wzorzec Dostępu do jak najmniejszej ilości informacji, ponieważ wywołanie nie ma innej możliwości niż zażądanie kompletnego rekordu danych według identyfikatora.

Tabela 4.1. Sposób, w jaki zastosowanie wzorca Dostępu do jak najmniejszej ilości informacji powoduje zmianę w projekcie interfejsu API

Interfejs API niezgodny ze wzorcem Dostępu do jak najmniejszej ilości informacji	Interfejs API zgodny ze wzorcem Dostępu do jak najmniejszej ilości informacji
`RequestCustomerData(id='12345')`	`RequestCustomerData(id='12345', items=['name', 'zip'])`
`{'id': '12345', 'name': 'Jane Doe', 'phone': '888-555-1212', 'zip': '01010', ...}`	`{"name": "Jane Doe", "zip": '01010'}`

Numer telefonu nie jest potrzebny, więc nie ma potrzeby, by interfejs go żądał; ale nawet jego zignorowanie zwiększa powierzchnię ataku dla napastnika próbującego go zdobyć. W prawej kolumnie znajduje się wersja tego samego interfejsu API, ale w wersji pozwalającej określić w wywołaniu, które pola są potrzebne (i tylko te są dostarczane), co minimalizuje przepływ prywatnych informacji.

Gdy weźmiemy pod uwagę również wzorzec **Bezpieczny z założenia** (ang. *secure by default*), domyślnym parametrem `items` powinien być minimalny zestaw pól, aby ograniczyć przepływ informacji (pod warunkiem że osoby dzwoniące będą mogły zażądać dokładnie tego, czego potrzebują).

Bezpieczny z założenia

Oprogramowanie powinno być zawsze bezpieczne zaraz „po wyjęciu z pudełka".

Projektuj oprogramowanie w taki sposób, aby *było bezpieczne z założenia* od samego początku, tzn. tak, aby brak działania ze strony osoby obsługującej nie stanowił zagrożenia. Dotyczy to zarówno ogólnej konfiguracji systemu, jak i opcji konfiguracyjnych komponentów i parametrów API. Bazy danych lub routery z domyślnymi hasłami notorycznie łamią ten schemat, ale do dziś ta wada projektowa jest zaskakująco często spotykana.

Jeśli poważnie podchodzisz do kwestii bezpieczeństwa, nigdy nie konfiguruj w urządzeniu potencjalnie niebezpiecznych ustawień z zamiarem zabezpieczenia ich później, ponieważ tworzysz w ten sposób potencjalne podatności na ataki (a zbyt często zapomina się o zabezpieczeniu). Jeśli np. musisz używać sprzętu z domyślnym hasłem przed podłączeniem go do ogólnodostępnej sieci, najpierw skonfiguruj go bezpiecznie w sieci prywatnej za zaporą ogniową. Pionierem w tej dziedzinie jest stan Kalifornia, w którym prawnie nakazano stosowanie tego wzorca (ustawa senacka nr 327 (2018) zakazuje stosowania domyślnych haseł w podłączonych do sieci urządzeniach).

Zasada Bezpieczny z założenia odnosi się do każdego ustawienia lub konfiguracji, które mogą mieć destrukcyjny wpływ na bezpieczeństwo, a nie tylko w odniesieniu do domyślnych haseł. Uprawnienia powinny być domyślnie ustawione na bardziej restrykcyjne. Użytkownicy powinni być zmuszeni do jawnej zmiany tych ustawień na mniej restrykcyjne, jeśli jest to konieczne i to tylko wtedy, gdy jest to bezpieczne. Należy domyślnie wyłączyć wszystkie potencjalnie niebezpieczne

opcje. Z drugiej strony, domyślnie trzeba włączać funkcje, które mają wpływ na bezpieczeństwo, aby działały od samego początku. Co oczywiste, ważne jest, by instalować aktualizacje oprogramowania: nie należy zaczynać pracy ze starą wersją oprogramowania (być może zawierającą znane luki w zabezpieczeniach) i mieć nadzieję, że w pewnym momencie zostanie ona zaktualizowana.

W idealnej sytuacji nie powinieneś nigdy mieć włączonych niebezpiecznych opcji konfiguracyjnych. Należy starannie rozważyć proponowane opcje konfiguracyjne, ponieważ łatwo można ustawić niebezpieczną opcję, która później stanie się pułapką dla innych. Należy również pamiętać, że każda nowa opcja zwiększa liczbę możliwych kombinacji, a zadanie polegające na sprawdzeniu, że wszystkie kombinacje tych ustawień są użyteczne i bezpieczne, staje się coraz trudniejsze wraz ze wzrostem liczby opcji. Jeśli musisz włączyć niebezpieczną konfigurację, postaraj się zawczasu wyjaśnić administratorowi, na czym polega ryzyko.

Jednak wzorzec Bezpieczny z założenia ma znacznie szersze zastosowanie niż tylko w odniesieniu do opcji konfiguracyjnych. Również w przypadku nieokreślonych parametrów API domyślne ustawienia dla nich powinny być bezpieczne. Przeglądarka akceptująca adres URL wpisany do paska adresu bez podania protokołu powinna zakładać, że witryna używa HTTPS i powraca do HTTP tylko wtedy, gdy nie uda się nawiązać połączenia HTTPS. Dwa równorzędne serwery negocjujące nowe połączenie HTTPS powinny domyślnie w pierwszej kolejności akceptować bezpieczniejszy zestaw szyfrów.

Listy dozwolonych zamiast List zabronionych

Przy projektowaniu mechanizmów bezpieczeństwa należy preferować **Listy dozwolonych** (ang. *allowlist*) zamiast **List zabronionych** (ang. *blocklist*). Listy dozwolonych to wyliczenie wszystkiego tego, co jest bezpieczne, więc z natury są skończone. Z kolei tworząc Listę zabronionych (listę blokad), próbujesz umieścić na niej wszystko to, co nie jest bezpieczne. W ten sposób potencjalnie możesz mieć nieskończony zestaw wszystkiego, co do czego *możesz mieć nadzieję*, że jest bezpieczne. Nie ulega wątpliwości, które podejście jest bardziej ryzykowne.

Na początek podam przykład niezwiązany z oprogramowaniem, aby upewnić się, że rozumiesz, co oznacza alternatywa między listą dozwolonych a listą zabronionych i dlaczego zawsze należy stosować listy dozwolonych. W pierwszych miesiącach obowiązywania nakazu pozostawania w domu z powodu COVID-19 gubernator mojego stanu nakazał zamknięcie plaż z następującymi zastrzeżeniami (przedstawionymi tutaj w uproszczonej formie):

> Zabrania się siadania, stania, kładzenia się, wylegiwania się, opalania i przebywania na plaży z wyjątkiem: „biegania, joggingu lub spacerowania po plaży, o ile zachowane są wymogi dotyczące zachowania dystansu społecznego" (dozwolone jest również przechodzenie przez plażę w celu surfowania).

Pierwsza klauzula jest listą zabronionych, ponieważ wymienia, jakie działania są niedozwolone, a druga klauzula wyjątków jest listą dozwolonych, gdyż zezwala na działania wymienione na liście. Ze względu na kwestie prawne może istnieć uzasadnienie dla sformułowania informacji w takiej formie, ale ze ściśle logicznego punktu widzenia uważam, że pozostawia ona wiele do życzenia.

Najpierw zastanówmy się nad listą zabronionych. Jestem przekonany, że istnieją inne ryzykowne czynności, które ludzie mogą wykonywać na plaży, a których nie zabrania pierwsza klauzula. Jeśli intencją nakazu było zapewnienie ludziom ruchu, to pominięto wiele z nich — np. klęczenie, a także jogę i występy żywych posągów. Problem z listami zabronionych polega na tym, że wszelkie pominięcia stają się wadami. Jeśli nie potrafisz wymienić wszystkich możliwych złych przypadków, system nie będzie bezpieczny.

Rozważmy teraz listę dozwolonych zajęć na plaży. Chociaż ona również jest niekompletna — bo kto by zakwestionował, że skakanie też jest w porządku — nie spowoduje to dużego problemu z bezpieczeństwem. Być może ułamek procenta plażowiczów zostanie niesprawiedliwie ukarany, a szkoda jest niewielka, Co ważniejsze, niekompletna lista nie spowoduje powstania luki, która umożliwi wykonywanie ryzykownych czynności. Dodatkowe bezpieczne elementy, które początkowo zostały pominięte, można łatwo dodać do listy dozwolonych.

Mówiąc bardziej ogólnie, należy myśleć o kontinuum — od niedozwolonego po lewej stronie i, stopniowo, do dozwolonego po prawej. Gdzieś pośrodku znajduje się linia podziału. Celem jest dopuszczenie dobrych rzeczy po prawej stronie linii a niedopuszczenie do złych po lewej. Listy dozwolonych tworzą linię z prawej strony, a następnie stopniowo przesuwają ją w lewo, włączając kolejne elementy kontinuum, w miarę jak rośnie lista tego, co można dopuścić. Jeśli pominiesz coś dobrego na liście dozwolonych rzeczy, nadal będziesz po bezpiecznej stronie nieuchwytnej linii, która jest prawdziwym podziałem. Być może nigdy nie dotrzesz do punktu, w którym wszystkie bezpieczne działania będą dozwolone, a wtedy każdy dodatek do listy będzie zbyteczny, ale dzięki tej technice łatwo pozostać po bezpiecznej stronie. Porównaj to z podejściem opartym na liście blokad: jeśli nie wyliczymy wszystkiego na lewo od prawdziwego podziału, dopuścimy coś, co dopuszczone być nie powinno. Najbezpieczniejszą listą blokad będzie taka, która zawiera prawie wszystko, co może być zbyt restrykcyjne, więc tak czy inaczej nie działa dobrze.

Często użycie Listy dozwolonych jest tak oczywiste, że nie postrzegamy tego jako wzorca. Przykładowo całkiem racjonalne jest, że niewielka grupa zaufanych menedżerów banku jest upoważniona do zatwierdzania transakcji o dużej wartości. Nikomu nie przyszłoby do głowy przygotowanie listy blokad złożonej z nazwisk wszystkich pracowników, którzy *nie są upoważnieni* do zatwierdzania transakcji (co milcząco pozwoliłoby każdemu nieznajdującemu się na liście pracownikowi na taki przywilej). Jednak niedbali programiści mogą próbować weryfikacji poprawności danych wejściowych, sprawdzając, czy wartość nie zawiera żadnego ze znajdujących się na liście nieprawidłowych znaków, a przy takim podejściu łatwo zapomnieć o takich znakach jak `NULL` (ASCII 0) lub `DEL` (ASCII 127).

Jak na ironię, prawdopodobnie najlepiej sprzedający się produkt związany z bezpieczeństwem — oprogramowanie antywirusowe — próbuje blokować wszystkie znane złośliwe programy. Nowoczesne produkty antywirusowe są znacznie bardziej zaawansowane niż stare wersje, które polegały na porównywaniu skrótów z bazą danych znanego złośliwego oprogramowania, ale mimo to wydaje się, że wszystkie one w pewnym stopniu działają w oparciu o listę blokad. (Jest to też wspaniały przykład bezpieczeństwa przez zaciemnienie — większość komercyjnego oprogramowania antywirusowego jest własnościowa — więc możemy tylko spekulować). Co prawda, ma sens to, że zdecydowano się na techniki oparte na listach zabronionych: wiadomo, jak gromadzić przykładowe złośliwe oprogramowanie, ale perspektywa utworzenia listy wszystkich dobrych programów na świecie przed wydaniem kolejnej wersji oprogramowania antywirusowego wydaje się być nie do zrealizowania. Nie chodzi mi o żaden konkretny produkt ani o ocenę jego wartości, ale o wybór sposobu ochrony za pomocą Listy zabronionych i o to, dlaczego jest to bardzo ryzykowne.

Unikaj przewidywalności

Wszelkie dane (lub zachowania), które są przewidywalne, nie mogą być utrzymywane w tajemnicy, ponieważ napastnicy mogą je poznać poprzez zgadywanie.

Przewidywalność danych w projektowaniu oprogramowania może prowadzić do poważnych błędów, ponieważ może skutkować wyciekiem informacji. Rozważmy prosty przykład przypisywania identyfikatorów kont nowych klientów. Gdy nowy klient rejestruje się na stronie internetowej, system potrzebuje unikalnego identyfikatora, aby oznaczyć konto. Jednym z oczywistych i łatwych sposobów jest nazwanie pierwszego konta 1, drugiego 2 itd. To działa, ale co zdradza z punktu widzenia atakującego?

Nowe identyfikatory kont zapewniają teraz atakującemu łatwy sposób na poznanie liczby kont użytkowników utworzonych do tej pory. Jeśli np. atakujący będzie okresowo tworzył nowe konto, będzie miał dokładną informację o tym, ile kont klientów witryna posiada w danym momencie — czyli informację, którą większość firm niechętnie ujawniłaby konkurentowi. W zależności od specyfiki systemu możliwych jest wiele innych pułapek. Inną konsekwencją złego projektu jest to, że osoby atakujące mogą łatwo odgadnąć identyfikator konta przypisany do kolejnego nowo utworzonego konta, a uzbrojone w tę wiedzę mogą być w stanie przeszkodzić w konfiguracji nowego konta, przywłaszczając je sobie i oszukując system rejestracji.

Problem przewidywalności przybiera wiele postaci, w przypadku odmiennych projektów możemy mieć do czynienia z różnego rodzaju wyciekami. Przykładowo identyfikator konta, który zawiera kilka liter nazwiska lub kodu pocztowego właściciela konta, niepotrzebnie byłby wskazówką dotyczącą jego tożsamości. Oczywiście ten sam problem dotyczy identyfikatorów stron internetowych, wydarzeń i innych. Najprostszy środek łagodzący w tym wypadku jest taki, że jeśli identyfikator ma być unikalnym pseudonimem, to powinien właśnie takim być — nigdy nie może być liczbą użytkowników, adresem e-mail użytkownika ani bazować na innych informacjach umożliwiających identyfikację.

Prostym sposobem uniknięcia tych problemów jest używanie *bezpiecznie losowych* identyfikatorów. Prawdziwie losowe wartości nie są łatwe do odgadnięcia, więc nie powodują wycieku informacji. (Ściśle rzecz biorąc, długość identyfikatorów ujawnia maksymalną liczbę możliwych identyfikatorów, ale zazwyczaj nie są to informacje wrażliwe). Generatory liczb losowych będące standardowym narzędziem systemowym występują w dwóch odmianach: są to generatory liczb pseudolosowych i bezpieczne generatory liczb losowych (działają one wolniej). Powinieneś używać generatora w opcji bezpiecznej, chyba że jesteś pewien, że przewidywalność generatora nie jest szkodliwa. Więcej informacji na temat bezpiecznych generatorów liczb losowych można znaleźć w rozdziale 5.

Bezpieczna awaria

Jeśli wystąpi problem, należy upewnić się, że nie pozostawi nas w stanie zagrożenia.

W świecie fizycznym ten wzorzec jest zdroworozsądkowy. Doskonałym przykładem jest bezpiecznik elektryczny starego typu: jeśli przepływa przez niego zbyt duży prąd, ciepło topi metal, otwierając obwód. Prawa fizyki uniemożliwiają uszkodzenie bezpiecznika w taki sposób, aby pomimo uszkodzenia dopuścił do nadmiernego przepływu prądu. Ten przykład może wydawać się oczywisty, ale ponieważ oprogramowanie jest takie jakie jest (nie mamy praw fizyki po swojej stronie), łatwo go zlekceważyć.

Wiele zadań związanych z kodowaniem oprogramowania, które na początku wydają się niemal trywialne, staje się coraz bardziej skomplikowanych z powodu obsługi błędów. Normalnie przepływ programu może być prosty, ale gdy połączenie zostanie zerwane, alokacja pamięci nie powiedzie się, dane wejściowe są nieprawidłowe lub pojawia się wiele innych potencjalnych problemów, kod musi kontynuować działanie, jeśli to możliwe, lub elegancko się wycofać. Podczas pisania kodu można odnieść wrażenie, że spędza się więcej czasu na zajmowaniu się tymi wszystkimi rozproszeniami niż na wykonywaniu zadania. Łatwo jest szybko odrzucić kod obsługujący błędy jako nieistotny, co powoduje, że jest on często źródłem luk. Atakujący, jeśli tylko mogą, celowo wywołują takie przypadki błędów w nadziei, że istnieje luka, którą mogą wykorzystać.

Dokładne testowanie przypadków błędów często jest żmudne, zwłaszcza gdy kombinacje wielu błędów mogą się łączyć w nowe ścieżki kodu, więc może to być podatny grunt do ataku. Upewnij się, że każdy błąd jest albo bezpiecznie obsługiwany, albo prowadzi do całkowitego odrzucenia żądania. Gdy np. ktoś przesyła zdjęcie do serwisu udostępniającego zdjęcia, należy natychmiast sprawdzić, czy ma ono odpowiedni format (zdjęcia w niewłaściwym formacie są często wykorzystywane w złośliwy sposób), a jeśli nie ma, należy natychmiast usunąć je z pamięci masowej, aby uniemożliwić jego dalsze wykorzystanie.

Zdecydowane egzekwowanie reguł

W tych wzorcach chodzi o to, jak zapewnić, że zachowanie kodu będzie wymuszało dokładne egzekwowanie reguł. Kruczki prawne są zmorą każdego prawa i przepisów. Wzorce te pokazują, jak zapobiegać pisaniu kodu stwarzającego okazję do omijania systemu. Zamiast pisać kod i uzasadniać, że „nie sądzisz, by to tak zadziałało", lepiej zaprojektować go w taki sposób, aby zabronione operacje nie mogły zostać wykonane.

Pełna mediacja

Chroń wszystkie ścieżki dostępu, wymuszając taki sam dostęp bez wyjątku.

Niejasny termin dla oczywistej idei — **Pełna mediacja** (ang. *complete mediation*) — oznacza bezpieczne, spójne sprawdzanie wszystkich metod dostępu do chronionego zasobu. Jeśli istnieje wiele metod dostępu do zasobu, wszystkie one muszą być poddawane takiej samej kontroli autoryzacji, bez możliwości ich obejścia w sposób bardziej swobodny lub według bardziej liberalnych zasad.

Załóżmy np., że polityka systemu informatycznego firmy zajmującej się inwestycjami finansowymi mówi, iż zwykli pracownicy nie mogą sprawdzać identyfikatorów podatkowych klientów bez zgody kierownika, więc system udostępnia im ograniczony widok rekordów klientów z pominięciem tego pola. Menedżerowie mogą uzyskać dostęp do pełnego rekordu, a w rzadkich przypadkach, gdy osoba niebędąca menedżerem z uzasadnionych przyczyn potrzebuje tego identyfikatora, może poprosić menedżera o sprawdzenie danych. Pracownicy pomagają klientom na wiele sposobów, a jednym z nich jest dostarczanie kopii dokumentów podatkowych (jeśli z jakiegoś powodu klienci nie otrzymali ich pocztą). Po potwierdzeniu tożsamości klienta pracownik prosi o duplikat formularza (PDF), który drukuje i wysyła do klienta. Problem z tym systemem polega na tym, że na formularzu podatkowym pojawia się numer identyfikacji podatkowej klienta, do którego pracownik nie powinien mieć dostępu. Jest to błąd podpadający pod wzorzec Pełnej mediacji. Nieuczciwy pracownik mógłby poprosić o formularz podatkowy dowolnego klienta tak, jakby prosił o kopię, tylko po to, aby poznać jego numer identyfikacji podatkowej, co jest sprzeczne z zasadami zapobiegającymi ujawnianiu informacji pracownikom.

Najlepszym sposobem przestrzegania tego wzorca jest — jeśli to tylko możliwe — posiadanie jednego punktu, w którym podejmowane są określone decyzje dotyczące bezpieczeństwa. Często jest on nazywany **strażnikiem** (ang. *guard*) lub nieformalnie **wąskim gardłem** (ang. *bottleneck*). Chodzi o to, aby wszystkie wejścia do danego zasobu musiały przejść przez jedną bramkę. Alternatywnie, jeśli jest to niewykonalne, a wiele ścieżek wymaga strażników, wówczas wszystkie kontrole dla tego samego rodzaju dostępu powinny być funkcjonalnie równoważne, a najlepiej zaimplementowane jako identyczny kod.

W praktyce konsekwentna realizacja tego schematu może być trudna. Istnieją różne stopnie zgodności w zależności od zastosowanych zabezpieczeń:

Wysoka zgodność

Dostęp do zasobów dozwolony jest tylko przez jedną wspólną procedurę (strażnik wąskiego gardła).

Średnia zgodność

Dostęp do zasobów w różnych miejscach, z których każde jest chronione przez identyczną kontrolę autoryzacji (wspólny zwielokrotniony strażnik).

Niska zgodność

Dostęp do zasobów w różnych miejscach różnie strzeżonych przez niespójne kontrole autoryzacji (niepełna mediacja).

Kontrprzykład pokazuje, dlaczego projekty z prostymi politykami autoryzacji, w których skoncentrowano sprawdzanie autoryzacji w jednej wąskiej ścieżce kodu dla danego zasobu, są najlepszym sposobem na poprawną realizację tego wzorca. Użytkownik serwisu Reddit opisał niedawno, jak łatwo może dojść do pomyłki.

> Widziałem, że moja 8-letnia siostra korzystała na swoim telefonie iPhone 6 z systemem iOS 12.4.6 z serwisu YouTube po przekroczeniu limitu czasu ekranowego. Okazało się, że odkryła błąd związany z czasem ekranowym w wiadomościach, który pozwala użytkownikowi korzystać z aplikacji dostępnych w iMessage App Store.

Apple zaprojektował iMessage, tak aby zawierał własne aplikacje umożliwiające wywoływanie aplikacji YouTube na wiele sposobów. Jednak nie wprowadził kontroli czasu ekranowego na tej alternatywnej ścieżce do oglądania wideo — co jest klasycznym błędem Pełnej mediacji.

Należy unikać posiadania wielu ścieżek dostępu do tego samego zasobu, z których każda zawiera niestandardowy kod, potencjalnie działający nieco inaczej, ponieważ wszelkie rozbieżności mogą oznaczać słabsze działanie strażników na niektórych ze ścieżek. Stosowanie wielu strażników wymagałoby też wielokrotnego implementowania tego samego podstawowego sprawdzania i byłoby trudniejsze w utrzymaniu, ponieważ trzeba by było wprowadzać odpowiednie zmiany w kilku miejscach. Użycie wielu strażników wiąże się też z większym prawdopodobieństwem popełnienia błędu i większą ilością pracy przy dokładnym testowaniu.

Jak najmniej współdzielonych mechanizmów

Chodzi tu o utrzymanie izolacji między niezależnymi procesami za pomocą ograniczenia do minimum współdzielonych mechanizmów.

Aby jak najlepiej zrozumieć, co to oznacza i w czym pomaga, rozważmy przykład. Jądro systemu operacyjnego obsługującego wielu użytkowników zarządza zasobami systemowymi dla procesów działających w kontekstach różnych użytkowników. Zasadniczo konstrukcja jądra zapewnia izolację procesów, chyba że wyraźnie współdzielą one jakiś zasób lub kanał komunikacyjny. W sposób niewidoczny dla użytkownika jądro utrzymuje różne struktury danych niezbędne do obsługi żądań od wszystkich procesów użytkownika. Ten wzorzec zwraca uwagę

na to, że wspólny mechanizm struktur danych może nieumyślnie „zmostkować" (ang. *bridge*) procesy i dlatego najlepiej jest zminimalizować takie możliwości. Jeśli np. jakaś funkcjonalność może być zaimplementowana w kodzie środowiska użytkownika, gdzie granica procesu z konieczności izoluje ją od procesu, to prawdopodobieństwo, że funkcjonalność ta w jakiś sposób połączy procesy użytkowników, będzie mniejsze. W tym przypadku termin „zmostkowanie" oznacza wyciek informacji lub umożliwienie jednemu procesowi wpływania na inny bez autoryzacji.

Jeśli nadal wydaje Ci się to abstrakcyjne, rozważ następującą analogię, która nie dotyczy oprogramowania. Odwiedzasz swojego księgowego, aby przejrzeć zeznanie podatkowe na dzień przed upływem terminu składania dokumentów. Sterty papierów i segregatorów pokrywają biurko księgowego niczym miniaturowe drapacze chmur. Po przekopaniu się przez ten chaotyczny bałagan księgowy wyciąga Twoje dokumenty i rozpoczyna spotkanie. Gdy czekasz, możesz zobaczyć formularze podatkowe i wyciągi bankowe z nazwiskami i numerami identyfikacji podatkowej innych osób, ponieważ są na widoku. Być może Twój księgowy przez przypadek zapisze krótką notatkę na temat Twoich podatków na teczce innej osoby. Jest to dokładnie ten rodzaj pojawienia się połączenia (mostu) między niezależnymi stronami. Powstaje on, ponieważ księgowy używa biurka jako wspólnej przestrzeni roboczej — jest to sytuacja, której stara się uniknąć wzorzec Jak najmniej współdzielonych mechanizmów.

W następnym roku zatrudniasz innego księgowego, a kiedy się z nim spotykasz, wyjmuje z szafki Twoje akta. Otwiera je na swoim biurku, na którym panuje porządek i nie leżą dokumenty innych klientów. Tak właśnie działa mechanizm najmniejszego uprzywilejowania przy minimalnym ryzyku pomyłki lub wścibskich klientów widzących inne dokumenty.

W sferze oprogramowania można zastosować ten wzorzec, projektując usługi, które stanowią interfejs dla niezależnych procesów lub różnych użytkowników. Czy zamiast monolitycznej bazy danych, w której znajdują się dane wszystkich użytkowników, można zapewnić każdemu użytkownikowi oddzielną bazę danych lub w inny sposób ograniczyć dostęp do niej w zależności od kontekstu? Mogą istnieć dobre powody, aby umieścić wszystkie dane w jednym miejscu, ale jeśli zdecydujesz się nie stosować tego wzorca, bądź przygotowany na dodatkowe ryzyko i wymuś wprost niezbędną separację. Cookies są doskonałym przykładem zastosowania tego wzorca, ponieważ każdy klient niezależnie przechowuje swoje dane.

Nadmiarowość

Nadmiarowość (ang. *redundancy*) jest podstawową strategią bezpieczeństwa w inżynierii, która znajduje odzwierciedlenie w wielu zdroworozsądkowych praktykach, takich jak np. zapasowe opony w samochodach. Poniżej omówione wzorce pokazują, jak ją stosować, aby zwiększyć bezpieczeństwo oprogramowania.

Wielowarstwowa obrona

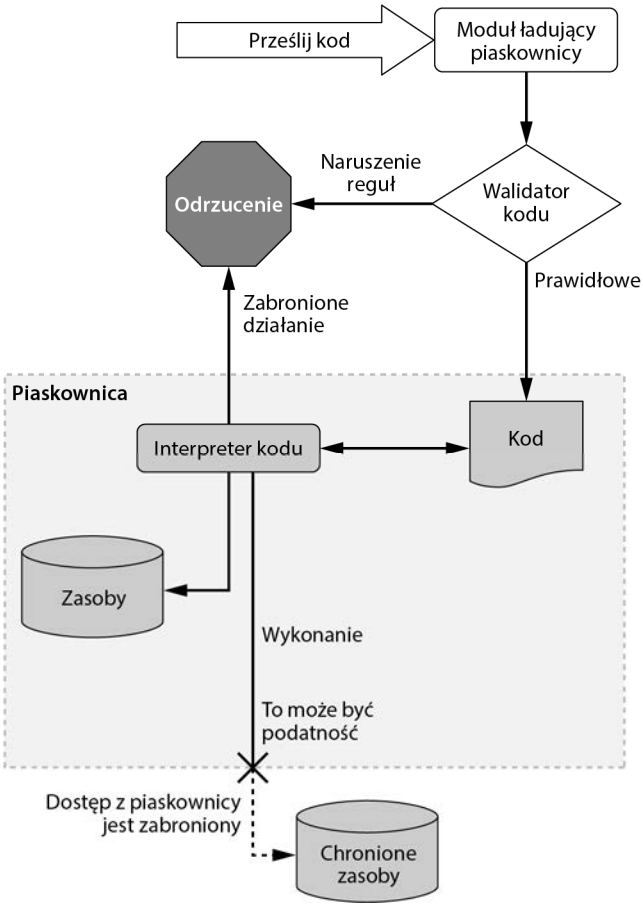
Połączenie niezależnych warstw ochrony zapewnia silniejszą ogólną obronę, która łącznie często jest bardziej skuteczna niż jakakolwiek pojedyncza warstwa.

Ta potężna technika jest jednym z najważniejszych wzorców, jakimi dysponujemy, aby nieuchronnie podatne na błędy systemy oprogramowania były bezpieczniejsze niż ich komponenty. Wyobraź sobie pokój, który chcesz przerobić na ciemnię fotograficzną, zasłaniając okno sklejką . Masz dużo sklejki, ale ktoś przypadkowo wywiercił kilka małych otworów w każdym arkuszu. Przybij tylko jeden arkusz, a liczne dziurki zepsują ciemność. Przybij drugi arkusz i o ile dwa otwory nie znajdą się przypadkiem w jednej linii, masz całkowicie ciemny pokój. Innym przykładem tego schematu jest punkt kontroli bezpieczeństwa, w którym stosuje się zarówno wykrywacz metalu, jak i kontrolę osobistą.

W dziedzinie projektowania oprogramowania należy wdrożyć mechanizm **Wielowarstwowej obrony** (ang. *defense in depth* — dosł. „obrona w głąb"), nakładając na siebie dwa lub więcej niezależnych mechanizmów ochrony w celu wymuszenia szczególnie ważnej decyzji dotyczącej bezpieczeństwa. Tak jak w przypadku dziurawej sklejki w każdej implementacji mogą występować wady, ale prawdopodobieństwo, że konkretny atak przeniknie przez obie, jest znikome, podobnie jak w przypadku dwóch otworów w sklejce, które przypadkowo znalazły się w jednej linii i przepuszczają światło. Ponieważ dwie niezależne weryfikacje wymagają podwójnego wysiłku i zajmują dwa razy więcej czasu, powinieneś używać tej techniki oszczędnie.

Doskonałym przykładem tej techniki, która równoważy wysiłek i koszty ogólne z korzyściami, jest implementacja **piaskownicy** (ang. *sandbox*), kontenera, w którym można bezpiecznie uruchamiać dowolny, niezaufany kod (nowoczesne przeglądarki internetowe uruchamiają WebAssembly w bezpiecznej piaskownicy). Uruchomienie niezaufanego kodu w systemie może mieć katastrofalne skutki, jeśli coś pójdzie nie tak, co jest uzasadnieniem nakładów poniesionych na ochronę wielowarstwową (rysunek 4.2).

Kod przeznaczony do wykonania w piaskownicy jest najpierw skanowany przez analizator (pierwsza warstwa obrony), który bada go pod kątem zestawu reguł. Jeżeli wystąpi jakiekolwiek naruszenie, system całkowicie odrzuca kod. Przykładowo jedna reguła może zabraniać wywołań do jądra, inna — używania określonych uprzywilejowanych instrukcji maszynowych. Kod może zostać załadowany do interpretera wtedy i tylko wtedy, gdy przejdzie to skanowanie. Interpreter uruchamiający kod jednocześnie egzekwuje szereg ograniczeń, które mają na celu zapobieganie wykonaniu uprzywilejowanych operacji tego samego typu. Aby atakujący mógł złamać ten system, musi najpierw ominąć sprawdzanie reguł przez skaner, a także zmusić interpreter do wykonania niedozwolonej operacji. Przykład ten jest szczególnie skuteczny, ponieważ skanowanie kodu i jego interpretacja różnią się zasadniczo, tak więc prawdopodobieństwo pojawienia się tego samego błędu w obu warstwach jest niewielkie, zwłaszcza jeśli są one opracowane niezależnie od siebie. Nawet jeśli istnieje szansa jeden do miliona, że skaner przeoczy konkretną technikę ataku — to samo dotyczy interpretera — to po połączeniu tych dwóch warstw cały system ma około jeden do tryliona (10^{18}) szans na rzeczywiste niepowodzenie. Na tym właśnie polega siła tego wzorca.



Rysunek 4.2. Przykład piaskownicy jako wzorca Wielowarstwowej obrony

Rozdzielanie przywilejów

Dwie strony są bardziej godne zaufania niż jedna.

Wzorzec **Rozdzielania przywilejów** (ang. *separation of priviledge*) znany jest również jako zasada **rozdzielenia obowiązków** (ang. *separation of duty*). Ma to odzwierciedlenie w niezaprzeczalnym fakcie, że dwa zamki trudniej sforsować niż jeden, gdy klucze do nich są powierzone dwóm różnym osobom (chociaż jest możliwe, że te dwie osoby mogą być w zmowie — co rzadko, ale się zdarza). Istnieją jeszcze inne dobre sposoby na zminimalizowanie tego ryzyka, ale w każdym razie jest to rozwiązanie o wiele lepsze niż poleganie wyłącznie na jednej osobie.

Przykładowo skrytki depozytowe są zaprojektowane w taki sposób, że bank odpowiada za bezpieczeństwo skarbca, w którym znajdują się wszystkie skrytki, a każdy posiadacz skrytki ma osobny klucz otwierający jego skrytkę. Bankowcy nie mogą dostać się do żadnej ze skrytek bez ich sforsowania np. przez wywiercenie zamków, a z kolei żaden z klientów nie zna kombinacji otwierającej sejf.

Dopiero wtedy, gdy klient uzyska dostęp do sejfu, a następnie użyje własnego klucza, może otworzyć swoją skrytkę.

Ten wzorzec należy stosować, gdy zakresy odpowiedzialności za chroniony zasób wyraźnie się pokrywają. Klasycznym przykładem jest zabezpieczanie centrum danych: w centrum danych jest administrator systemu z dostępem superużytkownika (lub jego zespół w przypadku dużych operacji) odpowiedzialny za obsługę maszyn. Dodatkowo są strażnicy kontrolujący fizyczny dostęp do obiektu. Taki odrębny zakres obowiązków, w połączeniu z odpowiednią kontrolą danych uwierzytelniających i kluczy dostępu powinien należeć do pracowników podlegających kierownictwom różnych pionów w organizacji. Zmniejsza to prawdopodobieństwo zmowy i zapobiega sytuacji, w której jeden z szefów wydałby polecenie wykonania nadzwyczajnego działania z naruszeniem protokołu. Szczególnie administratorzy pracujący zdalnie nie powinni mieć fizycznego dostępu do maszyn w centrum danych, a osoby przebywające fizycznie w centrum danych nie powinny znać żadnych kodów dostępu do maszyn ani kluczy potrzebnych do odszyfrowania jednostek pamięci masowej. Aby w pełni naruszyć bezpieczeństwo, konieczne byłoby współdziałanie dwóch osób (po jednej z każdej kontrolowanej domeny) w celu uzyskania dostępu zarówno fizycznego, jak i administracyjnego. W dużych organizacjach za różne zbiory danych zarządzane w centrum danych mogą odpowiadać różne grupy — co stanowi dodatkowy stopień separacji.

Innym zastosowaniem tego wzorca zwykle zarezerwowanym dla najbardziej krytycznych funkcji jest podzielenie odpowiedzialności pomiędzy wiele obowiązków (osób), aby uniknąć poważnych konsekwencji wynikających z błędu lub złych zamiarów jednego uczestnika. Jako dodatkowe zabezpieczenie przed ewentualnym wyciekiem kopii zapasowej danych można zaszyfrować ją dwukrotnie różnymi kluczami powierzonymi odrębnym osobom, aby można było z niej skorzystać tylko przy udziale obu stron. Ekstremalnym przykładem jest procedura uruchomienia pocisku jądrowego, co wymaga jednoczesnego przekręcenia dwóch kluczy w zamkach oddalonych od siebie o 3 metry, co gwarantuje, że żadna osoba działająca w pojedynkę nie będzie w stanie tego zrobić.

Zabezpiecz dzienniki zdarzeń za pomocą rozdzielenia uprawnień — jeden zespół będzie odpowiedzialny za rejestrowanie i przeglądanie zdarzeń, a drugi za pojawianie się zdarzeń. Oznacza to, że administratorzy mogą kontrolować aktywność użytkowników, ale oddzielna grupa musi kontrolować administratorów. W przeciwnym razie sprawca mógłby zablokować rejestrowanie własnych, szkodliwych działań lub manipulować dziennikiem zdarzeń w celu zatarcia śladów.

Nie da się osiągnąć rozdzielenia przywilejów na pojedynczym komputerze, ponieważ administrator z prawami superużytkownika ma na nim pełną kontrolę, ale nadal istnieje wiele sposobów na przybliżenie się do tego rozwiązania z dobrym skutkiem. Wdrożenie projektu z wieloma niezależnymi komponentami może być cennym środkiem łagodzącym, nawet jeśli administrator może go ostatecznie obejść: komplikuje to sabotaż; każdy atak trwa dłużej, jest bardziej prawdopodobne, że atakujący popełni błędy, co zwiększa prawdopodobieństwo złapania. Silne rozdzielenie przywilejów dla administratorów może polegać na wymuszeniu, by administrator pracował, korzystając ze specjalnej bramki `ssh` podlegającej

oddzielnej kontroli, w ramach której szczegółowo rejestrowana jest sesja i ewentualnie nałożone są inne ograniczenia.

Zagrożenia przychodzące z wewnątrz są trudne, a w niektórych przypadkach wręcz niemożliwe do wyeliminowania. Nie oznacza to, że środki zaradcze są stratą czasu. Sama świadomość tego, że ktoś nas obserwuje, jest już dużym czynnikiem odstraszającym. W takich środkach ostrożności chodzi nie tylko o brak zaufania: uczciwi pracownicy powinni z zadowoleniem przyjmować wszelkie rozwiązania typu Rozdzielenie przywilejów, ponieważ zwiększają możliwość rozliczenia z popełnionych działań i zmniejszają ryzyko będące efektem ich własnych błędów. Zmuszenie nieuczciwego pracownika do podjęcia wysiłku w celu zatarcia śladów spowalnia jego działania i zwiększa prawdopodobieństwo, że zostanie złapany na gorącym uczynku. Na szczęście ludzie mają dobrze rozwinięte systemy zaufania w bezpośrednich kontaktach ze współpracownikami, w związku z czym w praktyce nieuczciwe działania są niezwykle rzadkie.

Zaufanie i odpowiedzialność

Zaufanie i odpowiedzialność są spoiwem, które umożliwia współpracę. Systemy oprogramowania są w coraz większym stopniu połączone i współzależne — więc te wzorce są ważnymi drogowskazami.

Zasada ograniczonego zaufania

Zaufanie powinno być zawsze wyraźnym wyborem opartym na solidnych dowodach.

Wzorzec ten zwraca uwagę na to, że zaufanie jest cenne, ale zaleca sceptycyzm. Zanim istniało oprogramowanie, przestępcy wykorzystywali naturalną skłonność ludzi do ufania innym i aby uzyskać dostęp do pomieszczeń, przebierali się za robotników lub ludzi sprzedających cudowne leki; dopuszczali się też nieskończonej ilości innych oszustw. **Zasada ograniczonego zaufania** mówi, by nie zakładać, że osoba w mundurze musi być uczciwa i by brać pod uwagę, że dzwoniący, który twierdzi, że jest z FBI, może być oszustem. W przypadku oprogramowania wzorzec ten dotyczy sprawdzania autentyczności kodu przed jego zainstalowaniem oraz wymaga silnego uwierzytelnienia przed autoryzacją.

Doskonałym przykładem tego wzorca jest używanie plików cookie HTTP, co szczegółowo opisano w rozdziale 11. Serwery WWW umieszczają pliki cookie w odpowiedzi do klienta, oczekując, że klient odeśle je przy kolejnych żądaniach. Ponieważ jednak klienci nie mają obowiązku stosowania się do tych wymagań, serwery powinny zawsze traktować pliki cookie z przymrużeniem oka, a założenie, że klienci zawsze będą sumiennie wykonywać to zadanie, jest bardzo ryzykowne.

Zasada ograniczonego zaufania jest ważna, nawet w przypadku braku złych intencji. Przykładowo w systemie o krytycznym znaczeniu trzeba się upewnić, że wszystkie jego elementy spełniają takie same wysokie standardy jakości i bezpieczeństwa, aby nie narazić całego systemu na szwank. Błędne decyzje związane z zaufaniem — takie jak użycie kodu od nieznanego programisty (który

może zawierać złośliwe oprogramowanie lub błędy) — do wykonania krytycznej funkcji szybko osłabiają bezpieczeństwo. Ten wzorzec jest prosty i racjonalny, ale w praktyce może stanowić wyzwanie, ponieważ ludzie z natury są ufni, a stały brak zaufania może wywoływać poczucie paranoi.

Przyjmij odpowiedzialność za bezpieczeństwo

Wszyscy specjaliści zajmujący się oprogramowaniem mają jasny obowiązek przyjęcia odpowiedzialności za bezpieczeństwo; ta postawa powinna znaleźć odzwierciedlenie w tworzonym przez nich oprogramowaniu.

Przykładowo projektant powinien uwzględnić wymagania bezpieczeństwa podczas sprawdzania komponentów zewnętrznych, które mają zostać włączone do systemu. A na styku dwóch systemów obie strony powinny wyraźnie przyjąć na siebie określone obowiązki, których będą przestrzegać, a także potwierdzić wszelkie gwarancje, których dotrzymanie jest uzależnione od strony wywołującej.

Antywzorzec, który nie jest pożądany, zachodzi w sytuacji, w której pewnego dnia napotykamy problem i dwóch programistów mówi do siebie: „Myślałem, że ty zajmujesz się bezpieczeństwem, więc ja nie musiałem". W dużym systemie obie strony mogą łatwo znaleźć się w sytuacji, w której będą wskazywać na siebie nawzajem. Rozważmy sytuację, w której komponent A przyjmuje niezaufane dane wejściowe (np. jest to serwer frontend otrzymujący anonimowe żądanie z internetu) i przekazuje je (ewentualnie z pewnym przetworzeniem lub przeformatowaniem) do warstwy logiki biznesowej w komponencie B. Komponent A mógłby nie brać na siebie odpowiedzialności za bezpieczeństwo i ślepo przepuszczać wszystkie dane wejściowe, zakładając, że komponent B bezpiecznie obsłuży niezaufane dane wejściowe dzięki odpowiedniej walidacji i kontroli błędów. Z perspektywy komponentu B łatwo jest założyć, że serwer frontend wykona walidację wszystkich żądań i przekaże do B tylko te bezpieczne — więc B nie musi się już o to martwić. Właściwym sposobem radzenia sobie z tą sytuacją jest jasne uzgodnienie: należy podjąć decyzję, kto waliduje żądania i co jest zagwarantowane dla dalszych etapów przetwarzania (i czy w ogóle coś jest zapewnione). Aby uzyskać maksymalne bezpieczeństwo, należy zastosować mechanizm wielowarstwowej ochrony, w której oba komponenty niezależnie wykonają walidację danych wejściowych.

Rozważmy inny aż nazbyt często występujący przypadek, w którym pojawia się luka związana z odpowiedzialnością pomiędzy projektantem a użytkownikiem oprogramowania. Przypomnijmy sobie przykład ustawień konfiguracyjnych z naszej dyskusji na temat wzorca Bezpieczny z założenia, a konkretnie sytuację, w której pojawia się niebezpieczna opcja. Jeśli projektant wie, że dana opcja konfiguracyjna jest mniej bezpieczna, powinien poważnie rozważyć, czy udostępnienie tej opcji jest rzeczywiście konieczne. Oznacza to, że nie należy dostarczać użytkownikom opcji tylko dlatego, że można to łatwo zrobić, albo dlatego, że „ktoś, kiedyś może tego chcieć". Jest to równoznaczne z zastawieniem pułapki, w którą ktoś w końcu nieświadomie wpadnie. Jeśli istnieją uzasadnione powody dla potencjalnie ryzykownej konfiguracji, należy najpierw rozważyć metody zmiany

w projekcie, aby znaleźć bezpieczny sposób rozwiązania tego problemu. Niezależnie od tego, jeśli wymaganie jest z natury niebezpieczne, projektant powinien poinformować o tym użytkownika i chronić go przed konfigurowaniem tej opcji, gdy ten nie jest świadomy jej konsekwencji. Ważne jest nie tylko udokumentowanie ryzyka i zasugerowanie możliwych środków zaradczych, aby zniwelować podatność na zagrożenia. Użytkownicy powinni otrzymać jasną informację zwrotną — najlepiej by było to coś lepszego niż przerzucenie odpowiedzialności za pomocą wyświetlenia okna dialogowego „Czy jesteś pewien? (Dowiedz się więcej: *<łącze>*)".

CO JEST NIE TAK Z OKNEM DIALOGOWYM „CZY JESTEŚ PEWIEN"?

Uważam, że okna dialogowe „Czy jesteś pewien?" i im podobne są prawie zawsze błędem projektowym, który często zagraża bezpieczeństwu. Nie spotkałem się jeszcze z przykładem, w którym takie okno dialogowe byłoby najlepszym możliwym rozwiązaniem problemu. W przypadku konsekwencji związanych z bezpieczeństwem praktyka ta narusza wzorzec Wzięcia odpowiedzialności za bezpieczeństwo, ponieważ projektant zrzuca odpowiedzialność na użytkownika, który może nie być „pewien", ale naprawdę nie ma innego wyboru. Aby było jasne: te uwagi nie dotyczą normalnych potwierdzeń, takich jak interaktywne podpowiedzi polecenia `rm(1)` lub innych operacji, w których ważne jest uniknięcie przypadkowego działania.

Takie okna dialogowe mogą „paść ofiarą" zjawiska *zmęczenia oknami dialogowymi.* Ma ono miejsce, gdy osoby próbujące coś zrobić odruchowo odrzucają okna dialogowe, niemal powszechnie uważając je za przeszkodę, a nie pomoc. Ja sam (choć jestem świadomy bezpieczeństwa) w obliczu takich okien dialogowych również zastanawiam się: „Jak inaczej mogę zrobić to, co chcę zrobić?". Mam do wyboru albo zrezygnować z tego, co chcę zrobić, albo działać na własne ryzyko — i tylko mogę zgadywać, jakie to ryzyko, ponieważ nawet jeśli pojawia się tekst „dowiedz się więcej" — nigdy nie wydaje się, aby było to dobre rozwiązanie. W tym momencie pytanie „Czy na pewno?" sygnalizuje mi jedynie, że mam zamiar zrobić coś, czego będę potencjalnie żałować, bez dokładnego wyjaśnienia, co może się stać, i sugeruje, że prawdopodobnie nie ma już odwrotu.

Chciałbym, aby w tych oknach dialogowych dodawano trzecią opcję — „Nie, nie jestem pewien, ale i tak kontynuuj" — i aby była ona rejestrowana jako poważny błąd, ponieważ oprogramowanie zawiodło użytkownika. W każdej sytuacji, w której bezpieczeństwo ma krytyczne znaczenie, należy dokładnie przeanalizować przykłady tego rodzaju przerzucania odpowiedzialności i traktować je jako istotne błędy, które należy ostatecznie usunąć. Dokładny sposób ich eliminowania będzie zależał od konkretnych przypadków, ale istnieją pewne ogólne podejścia do przyjmowania odpowiedzialności. Należy jasno określić, co dokładnie ma się wydarzyć i dlaczego. Sformułowania powinny być zwięzłe, ale należy podać link lub równoważne odniesienie do pełnego wyjaśnienia i dobrej dokumentacji.Należy unikać niejasnych sformułowań („Czy na pewno chcesz to zrobić?") i pokazać dokładnie, jaki będzie cel działania (nie pozwól, aby okno dialogowe przesłoniło ważną informację). Nigdy nie używaj

podwójnego przeczenia lub mylących sformułowań („Czy na pewno chcesz się cofnąć?", gdzie odpowiedź „Nie" powoduje wybranie akcji). Jeśli to możliwe, udostępniaj opcję cofnięcia. Dobrym, coraz częściej spotykanym w dzisiejszych czasach wzorcem jest pasywne oferowanie możliwości cofnięcia wykonanego działania po każdej większej akcji. Jeśli nie ma możliwości cofnięcia, to w odpowiedniej dokumentacji należy zaproponować obejście tego problemu lub zasugerować wcześniejsze utworzenie kopii zapasowej danych. Należy dążyć do zmniejszania liczby takich „wyborów bez odwrotu", a najlepiej ograniczyć je i pozostawić do stosowania tylko dla profesjonalnych administratorów, którzy mają wiedzę i umiejętności pozwalające im przejąć odpowiedzialność.

Antywzorce

Naucz się widzieć w cudzym nieszczęściu kłopoty,
których powinieneś unikać.

— Publiusz Syrus

Niektórych umiejętności najlepiej nauczyć się, obserwując, jak pracuje mistrz, ale inny ważny rodzaj nauki wynika z unikania błędów popełnionych w przeszłości przez innych. Początkujący chemicy uczą się, że kwas należy zawsze rozcieńczać, dodając go do pojemnika z wodą — nigdy odwrotnie! Wiadomo, że w obecności dużej ilości kwasu pierwsza kropla wody reaguje gwałtownie, wytwarzając dużo ciepła, które może doprowadzić do zagotowania wody i jej pryskania wokół (wraz z kwasem). Chyba nikt nie chciałby samodzielnie przerabiać tej lekcji przez naśladowanie i w tym duchu przedstawiam tutaj kilka antywzorców, których najlepiej unikać, kiedy zajmujesz się bezpieczeństwem.

W poniższych krótkich punktach wymieniłem kilka antywzorców bezpieczeństwa oprogramowania. Wzorce te mogą generalnie nieść ryzyko dla bezpieczeństwa. Najlepiej ich unikać, choć właściwie nie są to podatności. W przeciwieństwie do wzorców opisanych wcześniej (mających ogólnie rozpoznawalne nazwy), tutaj dla tych, które nie mają dobrze ugruntowanych nazw, dla wygody wybrałem nazwy opisowe.

Reprezentant wprowadzony w błąd

Problem **Reprezentanta wprowadzonego w błąd** (ang. *confused deputy*) to podstawowe wyzwanie związane z bezpieczeństwem, które stanowi sedno wielu luk w oprogramowaniu. Można powiedzieć, że jest to ojciec wszystkich antywzorców. Aby wyjaśnić jego nazwę i znaczenie, dobrym punktem wyjścia będzie krótka historyjka. Przypuśćmy, że sędzia wydaje odpowiednim służbom nakaz aresztowania Normana Batesa. Osoba, która ma to wykonać, sprawdza, pod jakim adresem mieszka osoba „Norman Bates" i aresztuje mieszkającego tam mężczyznę. Mężczyzna upiera się, że to pomyłka, ale aresztujący słyszał już wcześniej takie

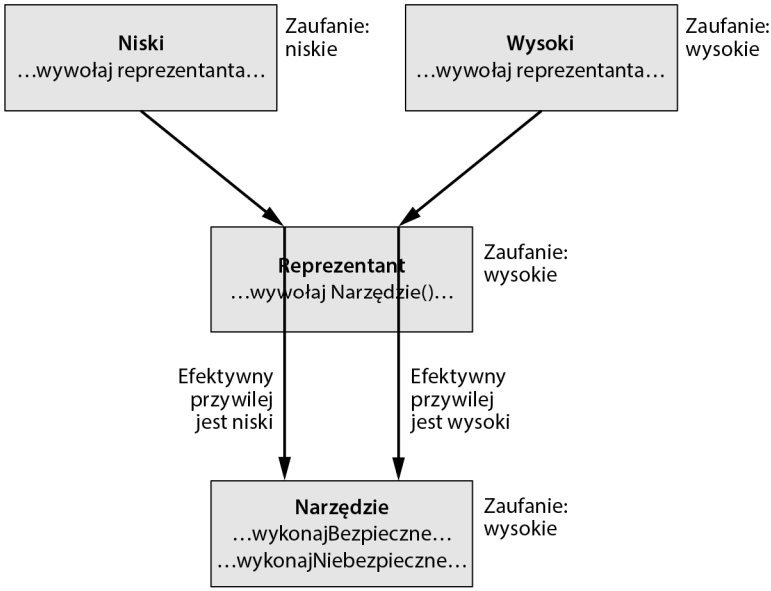
wymówki. Zwrot akcji w naszej opowieści (która nie ma nic wspólnego z *Psychozą*) powoduje fakt, że Norman, przewidując, że zostanie złapany, od lat posługuje się fałszywym adresem. Zastępca zdezorientowany tym podstępem niewłaściwie wykorzystał swoje uprawnienia do aresztowania — można powiedzieć, że Norman go wykorzystał, a właściwie przyznane mu uprawnienia do własnych, niecnych celów. (Doskonałym przykładem problemu reprezentanta wprowadzonego w błąd jest nikczemne przestępstwo *swattingu* — fałszywe zgłaszanie zagrożenia w celu skierowania sił policyjnych przeciwko niewinnym ofiarom — ale wolałbym nie opowiadać szczegółowo jednej z tych smutnych historii).

Typowe przykłady zdezorientowanych zastępców to jądro, gdy jest wywoływane przez kod środowiska użytkownika, lub serwer WWW, kiedy jest wywoływany z internetu. Byt wywołujący jest *reprezentantem* (ang. *deputy*), ponieważ kod o wyższych uprawnieniach jest wywoływany w celu wykonania czynności w imieniu bytu wywołującego o niższych uprawnieniach. Ryzyko to wynika bezpośrednio z przekroczenia granicy zaufania, dlatego też jest ono tak bardzo interesujące w modelowaniu zagrożeń. W kolejnych rozdziałach omówione zostaną liczne sposoby wprowadzania reprezentanta w błąd, w tym przepełnienia bufora, słaba walidacja danych wejściowych oraz ataki typu CSRF (ang. *cross-site request forgery*), by wymienić tylko kilka z nich. W przeciwieństwie do reprezentantów w ludzkiej postaci (którzy mogą polegać na instynkcie, dotychczasowym doświadczeniu i innych wskazówkach, w tym zdrowym rozsądku) oprogramowanie można łatwo zmusić do robienia rzeczy, do których nie było przeznaczone, chyba że zostało zaprojektowane i wdrożone z pełnym uwzględnieniem wszystkich niezbędnych środków ostrożności.

Intencje i złośliwość

Jak pisałem w rozdziale 1., aby oprogramowanie było godne zaufania, trzeba spełnić dwa warunki: musi być zbudowane przez ludzi, którym można ufać i są zarówno uczciwi, jak i kompetentni, by dostarczyć produkt wysokiej jakości. Tym, co różni te dwa warunki, są intencje. Problem z aresztowaniem Normana Batesa nie polegał na tym, że zastępca był nieuczciwy, ale na tym, że nie udało się prawidłowo zidentyfikować aresztowanego. Oczywiście kod nie jest nieposłuszny ani leniwy, ale źle napisany kod może łatwo działać w sposób inny niż zamierzony. Podczas gdy wielu łatwowiernych użytkowników komputerów, a czasami nawet technicznie uzdolnionych specjalistów obsługujących oprogramowanie, daje się nabrać na złośliwe oprogramowanie, wiele ataków polega na wykorzystaniu błędu w oprogramowaniu, które mimo że jest ze źródła godnego zaufania, to jest wadliwe.

Luki w zabezpieczeniach typu Reprezentant wprowadzony w błąd często powstają wtedy, gdy kontekst oryginalnego żądania zostaje utracony we wcześniejszej części kodu — np. gdy tożsamość żądającego nie jest już dostępna. Tego typu pomyłki są szczególnie prawdopodobne w kodzie, w którym wspólnie występują zarówno wywołania o wysokich, jak i niskich uprawnieniach. Rysunek 4.3 pokazuje, jak wygląda takie wywołanie.



Rysunek 4.3. Przykład antywzorca Reprezentant wprowadzony w błąd

Kod `Reprezentant` w centrum rysunku wykonuje zadania zarówno dla kodu o niskim, jak i wysokim poziomie uprawnień. Gdy jest wywoływany z poziomu `Wysoki` po prawej stronie, może wykonywać potencjalnie niebezpieczne operacje na rzecz swojego zaufanego rozmówcy. Wywołanie go z poziomu `Niski` oznacza przekroczenie granicy zaufania, dlatego `Reprezentant` powinien wykonywać tylko bezpieczne operacje, odpowiednie dla rozmówców z niskimi uprawnieniami. W ramach implementacji `Reprezentant` używa podkomponentu `Narzędzie` do wykonywania swojej pracy. Kod w podkomponencie `Narzędzie` nie rozróżnia rozmówców z wysokimi i niskimi uprawnieniami. Może więc błędnie wykonywać potencjalnie niebezpieczne operacje w imieniu *reprezentanta*, który nie powinien ich wykonywać dla wywołującego z niskimi przywilejami.

Reprezentant godny zaufania

Przeanalizujmy, jak być godnym zaufania reprezentantem, zaczynając od zastanowienia się, gdzie znajduje się niebezpieczeństwo. Przypomnijmy, że granice zaufania są miejscem, w którym pojawia się możliwość „zmylenia", ponieważ celem ataku typu Reprezentant wprowadzony w błąd jest wykorzystanie wyższych przywilejów reprezentanta. Tak długo jak reprezentant rozumie, o co chodzi i kto o to prosi, a także przeprowadza odpowiednie sprawdzenie autoryzacyjne, wszystko powinno być w porządku.

Przypomnij sobie poprzedni przykład z kodem `Reprezentant`, gdzie problem wystąpił w bazowym kodzie `Narzędzie`, który nie radził sobie z granicami zaufania, gdy był wywoływany z kodu o poziomie `Niski`. W pewnym sensie to kod `Reprezentant` nieświadomie sprawił, że program `Narzędzie` stał się zdezorientowanym

zastępcą. Jeśli `Narzędzie` nie było przeznaczone do obrony przed wywołaniami z niskimi uprawnieniami, to albo kod `Reprezentant` musi być dokładnie zabezpieczony przed byciem oszukanym, albo `Narzędzie` może wymagać modyfikacji, tak aby było świadome wywołań z niskimi uprawnieniami.

Innym typowym błędem, który może się zdarzyć, jest błąd związany z działaniami podejmowanymi w imieniu żądania. **Ukrywanie danych** (ang. *data hiding*) jest podstawowym wzorcem projektowym, w którym implementacja ukrywa wykorzystywane mechanizmy za abstrakcją, a reprezentant działa bezpośrednio na mechanizmie, choć sam żądający nie może tego robić. Przykładowo reprezentant może zapisywać informacje w dzienniku jako efekt uboczny żądania, ale żądający nie ma dostępu do tego dziennika. Gdy reprezentant dodaje wpis do dziennika, zgłaszający żądanie wykorzystuje jego przywileje, dlatego należy uważać na niezamierzone efekty uboczne. Jeżeli osoba zgłaszająca żądanie może przekazać reprezentantowi nieprawidłowo sformułowany ciąg znaków, który trafia do dziennika, powodując uszkodzenie danych i czyniąc je nieczytelnymi — jest to atak typu Reprezentant wprowadzony w błąd, który skutecznie niszczy wpisy dziennika. W takim przypadku obrona zaczyna się od zauważenia, że ciąg znaków od osoby składającej wniosek może trafić do dziennika, i biorąc pod uwagę jaki to może mieć potencjalny wpływ, wymaga np. sprawdzania poprawności danych wejściowych.

Wspomniany w rozdziale 3. model zabezpieczenia dostępu kodu (CAS) został zaprojektowany specjalnie po to, aby zapobiec powstawaniu podatności na ataki typu Reprezentant wprowadzony w błąd. Gdy kod o niskich uprawnieniach wywołuje reprezentujący go kod o wysokich uprawnieniach, efektywne uprawnienia są odpowiednio zmniejszane. Gdy reprezentant potrzebuje większych przywilejów, musi się ich jawnie domagać, potwierdzając, że pracuje „na prośbę" kodu o niższych uprawnieniach.

Podsumowując, można powiedzieć, że na granicach zaufania należy ostrożnie obchodzić się z danymi o niższym poziomie zaufania i wywołaniami o niższych uprawnieniach, aby nie stać się wprowadzonym w błąd reprezentantem. Konieczne jest zachowanie kontekstu związanego z żądaniami podczas wykonywania zadania, aby w razie potrzeby można było w pełni sprawdzić uprawnienia. Należy uważać, aby efekty uboczne nie pozwalały zlecającym na przekroczenie swoich uprawnień.

Przepływ zwrotny zaufania

Przepływ zwrotny zaufania (ang. *backflow of trust*) występuje zawsze wtedy, gdy element o niższym poziomie zaufania kontroluje element o wyższym poziomie zaufania. Przykładem tego jest sytuacja, w której administrator systemu używa swojego komputera osobistego do zdalnego administrowania systemem przedsiębiorstwa. Chociaż osoba ta ma odpowiednie uprawnienia i jest godna zaufania, jej domowy komputer nie należy do systemu przedsiębiorstwa i nie powinien być miejscem, z którego nawiązywana jest sesja z uprawnieniami administratora. W zasadzie można o tym myśleć, jak o strukturalnym podwyższeniu przywilejów, które tylko czeka na to, by je wykorzystać.

Choć w prawdziwym życiu nikt przy zdrowych zmysłach nie wpadłby na ten absurdalny schemat, w systemie informatycznym zaskakująco łatwo go przeoczyć. Należy pamiętać, że w tym przypadku nie liczy się zaufanie, jakim *obdarzamy* komponenty, ale to, na jakie zaufanie *zasługują* te komponenty. Modelowanie zagrożeń może ujawnić potencjalne problemy tego typu po dokładnym przyjrzeniu się granicom zaufania.

Haczyki innych firm

Inną formą antywzorca Przepływu zwrotnego zaufania jest sytuacja, w której haczyki w komponencie systemu zapewniają osobom trzecim dostęp, którego być nie powinno. Rozważmy krytyczny system biznesowy, który zawiera zastrzeżony komponent wykonujący jakiś specjalistyczny proces w systemie. Być może stosuje zaawansowaną sztuczną inteligencję (ang. *artificial intelligence*, AI) do przewidywania przyszłych trendów biznesowych, wykorzystując poufne dane dotyczące sprzedaży i codziennie aktualizując prognozy. Komponent AI jest nowatorski, dlatego firma, która go wytwarza, musi się nim codziennie zajmować. Aby działał jako system „pod klucz", potrzebuje bezpośredniego tunelu przez zaporę sieciową, żeby uzyskać dostęp do interfejsu administracyjnego.

Jest to również nienormalna relacja zaufania, ponieważ strona trzecia ma bezpośredni dostęp do serca systemu przedsiębiorstwa całkowicie poza zasięgiem administratorów. Jeśli dostawca AI byłby nieuczciwy lub zostałby zaatakowany, mógłby z łatwością przesłać na zewnątrz wewnętrzne dane firmy, a co gorsza — nie byłoby szans, aby się o tym dowiedzieć. Należy zauważyć, że w przypadku pewnej liczby haczyków problem ten może nie występować i być dopuszczalny. Jeżeli np. haczyk implementuje mechanizm automatycznej aktualizacji i jest zdolny tylko do pobierania i instalowania nowych wersji oprogramowania, to przy odpowiednim poziomie zaufania można to zaakceptować.

Komponenty, których nie da się załatać

Prawie zawsze jest tylko kwestią tego kiedy, a nie czy w ogóle ktoś odkryje lukę w popularnym komponencie. Gdy taka luka stanie się publicznie znana, o ile nie jest całkowicie odłączona od jakichkolwiek punktów ataku, należy ją natychmiast załatać. Każdy komponent w systemie, którego nie można załatać, w końcu stanie się stałym kłopotem.

Komponenty sprzętu z preinstalowanym oprogramowaniem często nie dają się załatać, ale ogólnie rzecz biorąc, niezależnie od intencji i zakresu działania do tej kategorii zaliczamy również każde oprogramowanie, które wydawca przestał wspierać lub zakończył działalność. W praktyce istnieje wiele kategorii oprogramowania, którego nie da się naprawić: niewspierane oprogramowanie dostarczane tylko w formie binarnej, kod zbudowany z wykorzystaniem przestarzałego kompilatora lub innej zależności, kod wycofany decyzją zarządu, kod uwikłany w proces sądowy, kod utracony w wyniku działania oprogramowania ransomware oraz —

co ciekawe — kod napisany w języku, takim jak COBOL, który jest tak stary, że obecnie brakuje doświadczonych programistów potrafiących coś w nim zrobić. Główni dostawcy systemów operacyjnych zazwyczaj zapewniają wsparcie i aktualizacje przez pewien okres czasu, po którym oprogramowanie staje się praktycznie nie do naprawienia. Nawet oprogramowanie, które można aktualizować, w praktyce może nie okazać się lepsze, jeśli jego producent nie zapewni terminowych wydań aktualizacji. Nie należy kusić losu, używając oprogramowania, do którego nie ma pewności, że będzie je można, w razie potrzeby, szybko zaktualizować.

UWAGA *Listę wzorców i antywzorców bezpiecznego projektowania przedstawionych w tym rozdziale można znaleźć w dodatku D.*


PROGRAM
PARTNERSKI
GRUPY 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
Helion

Troska o bezpieczeństwo jest najlepszą praktyką!

Bezpieczeństwo oprogramowania jest niezwykle ważnym i złożonym zagadnieniem. Proste i zawsze sprawdzające się zasady właściwie nie istnieją. Aby zapewnić systemom IT bezpieczeństwo, trzeba zacząć o nim myśleć już na wstępnym etapie projektowania oprogramowania i zaangażować w ten proces cały zespół, od najwyższego kierownictwa, przez architektów, projektantów, po testerów, a nawet przyszłych użytkowników systemu. Często się okazuje, że świadomość wagi problemów bezpieczeństwa jest w takim zespole niewielka, a wiedza — fragmentaryczna.

Ta książka powstała z myślą o architektach oprogramowania, projektantach, programistach i dyrektorach do spraw technicznych. Zwięźle i przystępnie opisano w niej, jak zadbać o bezpieczeństwo na wczesnym etapie projektowania oprogramowania i jak zaangażować w ten proces cały team. Najpierw zaprezentowano podstawowe pojęcia, takie jak zaufanie, zagrożenia, łagodzenie skutków, bezpieczne wzorce projektowe i kryptografia. Omówiono też szczegółowo proces tworzenia projektu oprogramowania i jego przegląd pod kątem bezpieczeństwa. Wyjaśniono, jakie błędy najczęściej pojawiają się podczas kodowania i w jaki sposób powodują powstawanie luk w zabezpieczeniach. Poszczególne zagadnienia zostały uzupełnione obszernymi fragmentami kodu w językach C i Python.

W książce:

- identyfikacja ważnych zasobów, obszarów ataku i granic zaufania w systemie
- ocena skuteczności różnych technik łagodzenia zagrożeń
- wzorce projektowe ułatwiające zapewnianie bezpieczeństwa
- podatności, w tym XSS, CSRF i błędy związane z pamięcią
- testy bezpieczeństwa
- ocena projektu oprogramowania pod kątem bezpieczeństwa

Loren Kohnfelder programuje od ponad pół wieku. Zajmował się wieloma dziedzinami programowania. W firmie Microsoft pracował nad zagadnieniami związanymi z bezpieczeństwem, przyczynił się do powstania pierwszej metodologii jego proaktywnego zapewniania. Ostatnio pracował w zespole do spraw prywatności w Google. Przeprowadził ponad sto przeglądów bezpieczeństwa projektów systemów komercyjnych o dużej skali.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9432-2
9 788328 394322
Cena: 79,00 zł
no starch press