Podstawowe mechanizmy obronne

Podstawowy problem związany z bezpieczeństwem aplikacji internetowych, polegający na tym, że żadne dane wprowadzane przez użytkownika nie są godne zaufania, powoduje powstanie wielu mechanizmów bezpieczeństwa, których aplikacje używają do obrony przed atakiem. Praktycznie wszystkie aplikacje wykorzystują mechanizmy, które są koncepcyjnie podobne, chociaż szczegóły projektu i skuteczność implementacji znacznie się różnią. Mechanizmy obronne stosowane przez aplikacje internetowe obejmują następujące podstawowe elementy:

* Obsługa dostępu użytkowników do danych i funkcji aplikacji, aby uniemożliwić użytkownikom uzyskanie nieautoryzowanego dostępu

* Obsługa danych wprowadzanych przez użytkownika w funkcjach aplikacji, aby zapobiec niepożądanym zachowaniom zniekształconych danych wejściowych

* Postępowanie z atakującymi w celu upewnienia się, że aplikacja zachowuje się odpowiednio, gdy jest bezpośrednim celem, podejmowanie odpowiednich środków obronnych i ofensywnych w celu udaremnienia atakującego

* Zarządzanie samą aplikacją poprzez umożliwienie administratorom monitorowania jej działań i konfigurowania jej funkcjonalności

Ze względu na ich kluczową rolę w rozwiązywaniu podstawowego problemu bezpieczeństwa, mechanizmy te stanowią również zdecydowaną większość powierzchni ataku typowej aplikacji. Jeśli znajomość wroga jest pierwszą zasadą wojny, to dogłębne zrozumienie tych mechanizmów jest głównym warunkiem wstępnym skutecznego atakowania aplikacji. Jeśli jesteś nowicjuszem w hakowaniu aplikacji internetowych (a nawet jeśli nie), powinieneś poświęcić trochę czasu, aby zrozumieć, jak działają te podstawowe mechanizmy w każdej napotkanej aplikacji i zidentyfikować słabe punkty, które narażają je na atak .

Obsługa dostępu użytkownika

Głównym wymogiem bezpieczeństwa, który musi spełniać praktycznie każda aplikacja, jest kontrolowanie dostępu użytkowników do jej danych i funkcjonalności. Typowa sytuacja obejmuje kilka różnych kategorii użytkowników, takich jak użytkownicy anonimowi, zwykli użytkownicy uwierzytelnieni i użytkownicy administracyjni. Ponadto w wielu sytuacjach różni użytkownicy mają dostęp do różnych zestawów danych. Na przykład użytkownicy aplikacji poczty internetowej powinni mieć możliwość czytania własnych wiadomości e-mail, ale nie wiadomości innych osób. Większość aplikacji internetowych obsługuje dostęp za pomocą trzech powiązanych ze sobą mechanizmów bezpieczeństwa:

* Uwierzytelnianie

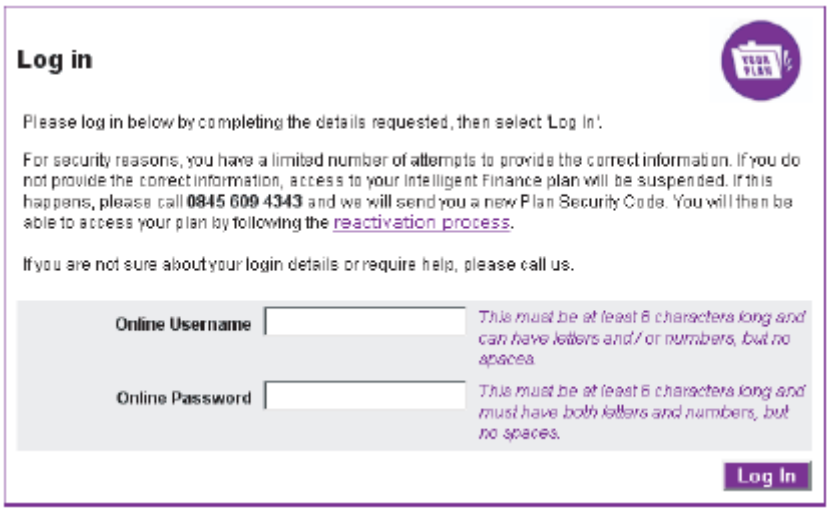
* Zarządzanie sesją

* Kontrola dostępu

Każdy z tych mechanizmów reprezentuje znaczący obszar powierzchni ataku aplikacji i każdy ma fundamentalne znaczenie dla ogólnego stanu bezpieczeństwa aplikacji. Ze względu na ich współzależności ogólne bezpieczeństwo zapewniane przez mechanizmy jest tak silne, jak najsłabsze ogniwo w łańcuchu. Wada dowolnego komponentu może umożliwić atakującemu uzyskanie nieograniczonego dostępu do funkcjonalności i danych aplikacji.

Uwierzytelnianie

Mechanizm uwierzytelniania jest logicznie najbardziej podstawową zależnością w obsłudze dostępu użytkownika przez aplikację. Uwierzytelnienie użytkownika polega na ustaleniu, że użytkownik jest w rzeczywistości tym, za kogo się podaje. Bez tej funkcji aplikacja musiałaby traktować wszystkich użytkowników jako anonimowych - z najniższym możliwym poziomem zaufania. Większość dzisiejszych aplikacji internetowych wykorzystuje konwencjonalny model uwierzytelniania, w którym użytkownik podaje nazwę użytkownika i hasło, które aplikacja sprawdza pod kątem ważności. Rysunek

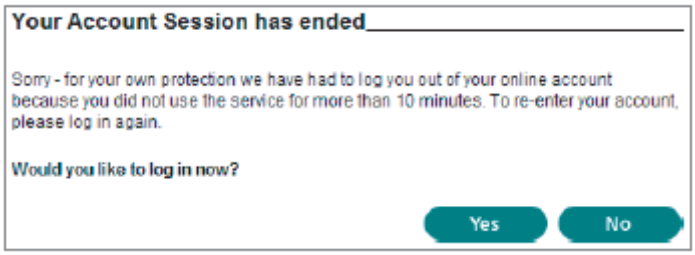


przedstawia typową funkcję logowania. W aplikacjach o krytycznym znaczeniu dla bezpieczeństwa, takich jak te używane przez banki internetowe, ten podstawowy model jest zwykle uzupełniany o dodatkowe dane uwierzytelniające i wieloetapowy proces logowania. Gdy wymagania bezpieczeństwa są jeszcze wyższe, można zastosować inne modele uwierzytelniania, oparte na certyfikatach klienta, kartach inteligentnych lub tokenach typu challenge-response. Oprócz podstawowego procesu logowania mechanizmy uwierzytelniania często wykorzystują szereg innych funkcji pomocniczych, takich jak samodzielna rejestracja, odzyskiwanie konta i funkcja zmiany hasła. Pomimo swojej powierzchownej prostoty, mechanizmy uwierzytelniania mają wiele wad zarówno projektowych, jak i implementacyjnych. Typowe problemy mogą umożliwić atakującemu zidentyfikowanie nazw użytkowników innych użytkowników, odgadnięcie ich haseł lub obejście funkcji logowania poprzez wykorzystanie defektów w jej logice. Kiedy atakujesz aplikację internetową, powinieneś zwrócić znaczną uwagę na różne funkcje związane z uwierzytelnianiem, które ona zawiera. Zaskakująco często wady tej funkcjonalności umożliwiają uzyskanie nieautoryzowanego dostępu do wrażliwych danych i funkcjonalności.

Zarządzanie sesją

Kolejnym logicznym zadaniem w procesie obsługi dostępu użytkownika jest zarządzanie sesją uwierzytelnionego użytkownika. Po pomyślnym zalogowaniu się do aplikacji, użytkownik uzyskuje dostęp do różnych stron i funkcji, wykonując serię żądań HTTP ze swojej przeglądarki. W tym samym czasie aplikacja otrzymuje niezliczoną ilość innych żądań od różnych użytkowników, z których część jest uwierzytelniona, a część anonimowa. Aby wymusić efektywną kontrolę dostępu, aplikacja potrzebuje sposobu na identyfikację i przetwarzanie serii żądań pochodzących od każdego unikalnego użytkownika. Praktycznie wszystkie aplikacje internetowe spełniają to wymaganie, tworząc sesję dla każdego użytkownika i wydawanie użytkownikowi tokena identyfikującego sesję. Sama sesja to zestaw struktur danych przechowywanych na serwerze, które śledzą stan interakcji użytkownika z aplikacją. Token to unikatowy ciąg, który aplikacja odwzorowuje na sesję. Gdy użytkownik otrzyma token, przeglądarka automatycznie przesyła go z powrotem do serwera w każdym kolejnym żądaniu HTTP,

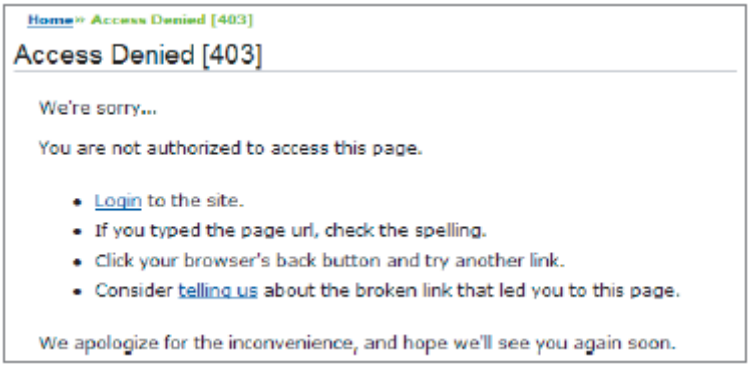
umożliwiając aplikacji powiązanie żądania z tym użytkownikiem. Pliki cookie HTTP są standardową metodą przesyłania tokenów sesji, chociaż wiele aplikacji używa do tego celu ukrytych pól formularza lub ciągu zapytania URL. Jeśli użytkownik nie wysyła żądania przez określony czas, w idealnym przypadku sesja wygasa, jak pokazano na rysunku



Jeśli chodzi o powierzchnię ataku, mechanizm zarządzania sesją jest w dużym stopniu zależny od bezpieczeństwa jego tokenów. Większość ataków przeciwko niemu ma na celu naruszenie bezpieczeństwa tokenów wydanych innym użytkownikom. Jeśli jest to możliwe, osoba atakująca może udawać użytkownika będącego ofiarą i używać aplikacji tak, jakby faktycznie uwierzytelniła się jako ten użytkownik. Główne obszary podatności wynikają z wad w sposobie generowania tokenów, które umożliwiają atakującemu odgadnięcie tokenów wydanych innym użytkownikom, oraz wad w sposobie późniejszej obsługi tokenów, co umożliwia atakującemu przechwycenie tokenów innych użytkowników. Niewielka liczba aplikacji eliminuje potrzebę stosowania tokenów sesji, korzystając z innych sposobów ponownej identyfikacji użytkowników w ramach wielu żądań. Jeśli używany jest wbudowany mechanizm uwierzytelniania HTTP, przeglądarka automatycznie przesyła poświadczenia użytkownika przy każdym żądaniu, umożliwiając aplikacji identyfikację użytkownika bezpośrednio na podstawie tych danych. W innych przypadkach aplikacja przechowuje informacje o stanie po stronie klienta, a nie na serwerze, zwykle w postaci zaszyfrowanej, aby zapobiec manipulacjom.

Kontrola dostępu

Ostatnim logicznym krokiem w procesie obsługi dostępu użytkowników jest podejmowanie i egzekwowanie właściwych decyzji dotyczących tego, czy każde indywidualne żądanie powinno być dozwolone, czy odrzucane. Jeśli opisane mechanizmy działają poprawnie, aplikacja zna tożsamość użytkownika, od którego otrzymuje każde żądanie. Na tej podstawie musi zdecydować, czy ten użytkownik jest upoważniony do wykonania czynności lub dostępu do danych, o które prosi, jak pokazano na rysunku .



Mechanizm kontroli dostępu zwykle musi implementować pewną szczegółową logikę, przy czym różne względy są istotne dla różnych obszarów aplikacji i różnych typów funkcjonalności. Aplikacja może obsługiwać wiele ról użytkowników, z których każda obejmuje różne kombinacje określonych

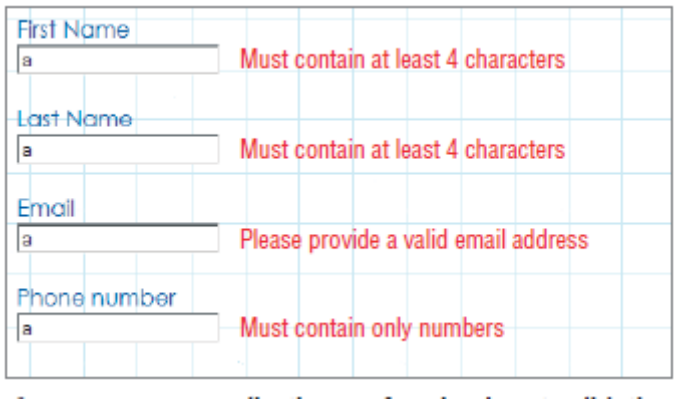
uprawnień. Poszczególnym użytkownikom można zezwolić na dostęp do podzbioru wszystkich danych przechowywanych w aplikacji. Określone funkcje mogą wprowadzać limity transakcji i inne kontrole, z których wszystkie muszą być odpowiednio egzekwowane w oparciu o tożsamość użytkownika. Ze względu na złożony charakter typowych wymagań kontroli dostępu mechanizm ten jest częstym źródłem luk w zabezpieczeniach, które umożliwiają atakującemu uzyskanie nieautoryzowanego dostępu do danych i funkcjonalności. Deweloperzy często dokonują błędnych założeń dotyczących interakcji użytkowników z aplikacją i często dokonują przeoczeń, pomijając kontrolę dostępu niektórych funkcji aplikacji. Wyszukiwanie tych luk jest często pracochłonne, ponieważ zasadniczo te same kontrole muszą być powtarzane dla każdego elementu funkcjonalności. Jednak ze względu na powszechność błędów w kontroli dostępu, ten wysiłek jest zawsze opłacalną inwestycją, gdy atakujesz aplikację internetową. Część 8 opisuje, w jaki sposób możesz zautomatyzować część wysiłku związanego z przeprowadzaniem rygorystycznych testów kontroli dostępu.

Obsługa danych wprowadzanych przez użytkownika

Przypomnij sobie podstawowy problem bezpieczeństwa opisany w Cczęści 1: Wszystkie dane wprowadzane przez użytkownika są niezaufane. Ogromna różnorodność ataków na aplikacje internetowe obejmuje wprowadzanie nieoczekiwanych danych wejściowych, spreparowanych w celu wywołania zachowania, które nie było zamierzone przez projektantów aplikacji. Odpowiednio, kluczowym wymaganiem dotyczącym zabezpieczeń aplikacji jest to, aby aplikacja obsługiwała dane wprowadzane przez użytkownika w bezpieczny sposób. Luki w zabezpieczeniach oparte na danych wejściowych mogą pojawić się w dowolnym miejscu w obrębie funkcjonalności aplikacji i w odniesieniu do praktycznie każdego rodzaju powszechnie używanej technologii. „Weryfikacja danych wejściowych" jest często wymieniana jako niezbędna obrona przed tymi atakami. Jednak żaden pojedynczy mechanizm ochronny nie może być zastosowany wszędzie, a obrona przed złośliwymi danymi wejściowymi często nie jest tak prosta, jak się wydaje.

Odmiany danych wejściowych

Typowa aplikacja internetowa przetwarza dane dostarczone przez użytkownika w wielu różnych formach. Niektóre rodzaje walidacji danych wejściowych mogą nie być wykonalne lub pożądane dla wszystkich tych form danych wejściowych. Rysunek przedstawia rodzaj sprawdzania poprawności danych wejściowych, często przeprowadzany przez funkcję rejestracji użytkownika.



W wielu przypadkach aplikacja może być w stanie nałożyć bardzo rygorystyczne kontrole poprawności na określony element danych wejściowych. Na przykład nazwa użytkownika przesyłana do funkcji logowania może wymagać maksymalnej długości ośmiu znaków i zawierać tylko znaki alfabetu. W innych przypadkach aplikacja musi tolerować szerszy zakres możliwych danych wejściowych. Na przykład pole adresowe przesłane na stronę z danymi osobowymi może zgodnie z prawem zawierać

litery, cyfry, spacje, łączniki, apostrofy i inne znaki. Jednak w przypadku tej pozycji nadal można nałożyć ograniczenia. Dane nie powinny przekraczać rozsądnego limitu długości (np. 50 znaków) i nie powinny zawierać żadnych znaczników HTML. W niektórych sytuacjach aplikacja może wymagać zaakceptowania dowolnych danych wejściowych od użytkowników. Na przykład użytkownik aplikacji do blogowania może utworzyć blog, którego tematem jest hakowanie aplikacji internetowych. Posty i komentarze na blogu mogą całkiem legalnie zawierać wyraźne ciągi ataku, które są omawiane. Aplikacja może wymagać zapisania tych danych wejściowych w bazie danych, zapisania ich na dysku i wyświetlenia użytkownikom w bezpieczny sposób. Nie może po prostu odrzucić danych wejściowych tylko dlatego, że wyglądają na potencjalnie złośliwe, bez znacznego zmniejszenia wartości aplikacji dla niektórych użytkowników. Oprócz różnego rodzaju danych wejściowych, które użytkownicy wprowadzają za pomocą interfejsu przeglądarki, typowa aplikacja otrzymuje wiele danych, które rozpoczęły swoje życie na serwerze i które są wysyłane do klienta, aby klient mógł przesłać je z powrotem do serwera na kolejne prośby. Obejmuje to elementy takie jak pliki cookie i ukryte pola formularzy, których zwykli użytkownicy aplikacji nie widzą, ale które atakujący może oczywiście przeglądać i modyfikować. W takich przypadkach aplikacje mogą często przeprowadzać bardzo szczegółowe sprawdzanie poprawności otrzymanych danych. Na przykład może być wymagane, aby parametr miał jedną z określonego zestawu znanych wartości, na przykład plik cookie wskazujący preferowany język użytkownika lub określony format, na przykład numer identyfikacyjny klienta. Ponadto, gdy aplikacja wykryje, że dane wygenerowane przez serwer zostały zmodyfikowane w sposób niemożliwy dla zwykłego użytkownika ze standardową przeglądarką, często oznacza to, że użytkownik próbuje zbadać aplikację pod kątem luk w zabezpieczeniach. W takich przypadkach aplikacja powinna odrzucić żądanie i zarejestrować incydent w celu potencjalnego zbadania.

Podejścia do obsługi danych wejściowych

Powszechnie przyjmuje się różne szerokie podejścia do problemu obsługi danych wprowadzanych przez użytkownika. Różne podejścia są często preferowane w różnych sytuacjach i różnych rodzajach danych wejściowych, a czasem pożądana może być kombinacja podejść.

„Odrzuć znane zło”

Takie podejście zwykle wykorzystuje czarną listę zawierającą zestaw dosłownych ciągów lub wzorców, o których wiadomo, że są wykorzystywane w atakach. Mechanizm sprawdzania poprawności blokuje wszelkie dane pasujące do czarnej listy i zezwala na wszystko inne. Ogólnie rzecz biorąc, jest to uważane za najmniej efektywne podejście do sprawdzania poprawności danych wprowadzonych przez użytkownika z dwóch głównych powodów. Po pierwsze, typową lukę w zabezpieczeniach aplikacji internetowej można wykorzystać przy użyciu szerokiej gamy danych wejściowych, które można zakodować lub przedstawić na różne sposoby. Z wyjątkiem najprostszych przypadków jest prawdopodobne, że czarna lista pominie niektóre wzorce danych wejściowych, które można wykorzystać do ataku na aplikację. Po drugie, techniki eksploatacji nieustannie ewoluują. Nowatorskie metody wykorzystywania istniejących kategorii luk raczej nie zostaną zablokowane przez obecne czarne listy. Wiele filtrów opartych na czarnej liście można ominąć z zawstydzającą łatwością, dokonując drobnych zmian w blokowanym wejściu. Na przykład:

* Jeśli SELECT jest zablokowany, wypróbuj SeLeCt

* Jeśli lub 1=1-- jest zablokowany, spróbuj lub 2=2--

* Jeśli alert('xss') jest zablokowany, wypróbuj prompt('xss')

W innych przypadkach filtry zaprojektowane do blokowania określonych słów kluczowych można ominąć, używając niestandardowych znaków między wyrażeniami, aby zakłócić tokenizację wykonywaną przez aplikację. Na przykład:

SELECT/*foo*/username,password/*foo*/FROM/*foo*/users

<img%09onerror=alert(1) src=a>

Wreszcie, liczne filtry oparte na czarnej liście, szczególnie te zaimplementowane w zaporach ogniowych aplikacji internetowych, były podatne na ataki bajtów NULL. Ze względu na różne sposoby obsługi łańcuchów w zarządzanych i niezarządzanych kontekstach wykonywania, wstawienie bajtu NULL w dowolnym miejscu przed zablokowanym wyrażeniem może spowodować, że niektóre filtry przestaną przetwarzać dane wejściowe i tym samym nie zidentyfikują wyrażenia. Na przykład:

%00<script>alert(1)</script>

UWAGA: Ataki wykorzystujące obsługę bajtów NULL pojawiają się w wielu obszarach bezpieczeństwa aplikacji internetowych. W kontekstach, w których bajt NULL działa jako ogranicznik łańcucha, można go użyć do zakończenia nazwy pliku lub zapytania do jakiegoś komponentu zaplecza. W kontekstach, w których bajty NULL są tolerowane i ignorowane (na przykład w HTML w niektórych przeglądarkach), dowolne bajty NULL można wstawić do zablokowanych wyrażeń, aby pokonać niektóre filtry oparte na czarnej liście.

„Akceptuj znane dobro”

To podejście wykorzystuje białą listę zawierającą zestaw dosłownych ciągów lub wzorców lub zestaw kryteriów, o których wiadomo, że pasują tylko do łagodnych danych wejściowych. Mechanizm sprawdzania poprawności zezwala na dane, które pasują do białej listy i blokuje wszystko inne. Na przykład przed wyszukaniem żądanego kodu produktu w bazie danych aplikacja może sprawdzić, czy zawiera on tylko znaki alfanumeryczne i ma dokładnie sześć znaków. Biorąc pod uwagę późniejsze przetwarzanie kodu produktu, programiści wiedzą, że dane wejściowe, które pomyślnie przejdą ten test, nie mogą powodować żadnych problemów. W przypadkach, w których takie podejście jest wykonalne, jest uważane za najskuteczniejszy sposób obsługi potencjalnie złośliwych danych wejściowych. Pod warunkiem, że podczas tworzenia białej listy zachowana zostanie należyta ostrożność, osoba atakująca nie będzie mogła użyć spreparowanych danych wejściowych do ingerencji w zachowanie aplikacji. Jednak w wielu sytuacjach aplikacja musi przyjąć do przetwarzania dane, które nie spełniają racjonalnych kryteriów tego, co określa się jako „dobre”. Na przykład imiona niektórych osób zawierają apostrof lub myślnik. Można ich używać w atakach na bazy danych, ale może być wymagane, aby aplikacja zezwalała każdemu na rejestrację pod swoim prawdziwym nazwiskiem. Dlatego, chociaż często jest to niezwykle skuteczne, podejście oparte na białej liście nie stanowi uniwersalnego rozwiązania problemu obsługi danych wprowadzanych przez użytkownika.

Sanityzacja

Podejście to uwzględnia potrzebę czasami akceptowania danych, których nie można zagwarantować jako bezpiecznych. Zamiast odrzucać te dane wejściowe, aplikacja oczyszcza je na różne sposoby, aby zapobiec ich negatywnym skutkom. Potencjalnie złośliwe znaki mogą zostać usunięte z danych, pozostawiając tylko to, co jest znane jako bezpieczne, lub mogą być odpowiednio zakodowane lub „ucieknięte” przed dalszym przetwarzaniem. Podejścia oparte na oczyszczaniu danych są często bardzo skuteczne iw wielu sytuacjach można na nich polegać jako na ogólnym rozwiązaniu problemu złośliwych danych wejściowych. Na przykład typową obroną przed atakami typu cross-site scripting jest kodowanie niebezpiecznych znaków w formacie HTML, zanim zostaną one osadzone na stronach

aplikacji. Skuteczne oczyszczanie może być jednak trudne do osiągnięcia, jeśli kilka rodzajów potencjalnie złośliwych danych musi zostać umieszczonych w jednym elemencie wejściowym. W tej sytuacji pożądane jest podejście do sprawdzania poprawności granic, jak opisano później.

Bezpieczna obsługa danych

Wiele luk w zabezpieczeniach aplikacji internetowych powstaje, ponieważ dane dostarczane przez użytkowników są przetwarzane w niebezpieczny sposób. Luk w zabezpieczeniach często można uniknąć, nie sprawdzając poprawności samych danych wejściowych, ale upewniając się, że przetwarzanie, które jest na nich wykonywane, jest z natury bezpieczne. W niektórych sytuacjach dostępne są bezpieczne metody programowania, które pozwalają uniknąć typowych problemów. Na przykład atakom typu SQL injection można zapobiegać poprzez prawidłowe stosowanie sparametryzowanych zapytań w celu uzyskania dostępu do bazy danych. W innych sytuacjach funkcjonalność aplikacji można zaprojektować w taki sposób, aby uniknąć z natury niebezpiecznych praktyk, takich jak przekazywanie danych wejściowych użytkownika do interpretera poleceń systemu operacyjnego. Tego podejścia nie można zastosować do każdego rodzaju zadań, które aplikacje internetowe muszą wykonywać. Ale tam, gdzie jest dostępny, jest skutecznym ogólnym podejściem do obsługi potencjalnie złośliwych danych wejściowych.

Kontrole semantyczne

Wszystkie opisane dotychczas mechanizmy obronne odpowiadają na potrzebę ochrony aplikacji przed różnego rodzaju zniekształconymi danymi, których treść została spreparowana w celu zakłócenia przetwarzania aplikacji. Jednak w przypadku niektórych luk dane wejściowe dostarczone przez osobę atakującą są identyczne z danymi wejściowymi, które może przesłać zwykły, niezłośliwy użytkownik. To, co czyni go złośliwym, to różne okoliczności, w których jest przesyłany. Na przykład osoba atakująca może próbować uzyskać dostęp do konta bankowego innego użytkownika poprzez zmianę numeru konta przesłanego w ukrytym polu formularza. Żadna ilość walidacji składni nie rozróżni danych użytkownika i atakującego. Aby zapobiec nieautoryzowanemu dostępowi, aplikacja musi zweryfikować, czy przesłany numer konta należy do użytkownika, który go podał.

Walidacja granicy

Pomysł sprawdzania poprawności danych poza granicami zaufania jest znany. Podstawowy problem związany z bezpieczeństwem aplikacji internetowych wynika z faktu, że dane otrzymane od użytkowników są niezaufane. Chociaż kontrole poprawności danych wejściowych zaimplementowane po stronie klienta mogą poprawić wydajność i wygodę użytkownika, nie dają żadnej pewności co do danych, które faktycznie docierają do serwera. Moment, w którym dane użytkownika są po raz pierwszy odbierane przez aplikację po stronie serwera, stanowi ogromną granicę zaufania. W tym momencie aplikacja musi podjąć środki, aby bronić się przed złośliwymi danymi wejściowymi. Biorąc pod uwagę charakter głównego problemu, kuszące jest myślenie o problemie sprawdzania poprawności danych wejściowych w kategoriach granicy między Internetem, który jest „zły” i niezaufany, a aplikacją po stronie serwera, która jest „dobra” i zaufana. Na tym obrazie rolą sprawdzania poprawności danych wejściowych jest oczyszczenie potencjalnie złośliwych danych po ich otrzymaniu, a następnie przekazanie czystych danych do zaufanej aplikacji. Od tego momentu dane mogą być zaufane i przetwarzane bez dalszych kontroli lub obaw o możliwe ataki. Jak stanie się oczywiste, gdy zaczniemy badać niektóre rzeczywiste luki w zabezpieczeniach, ten prosty obraz sprawdzania poprawności danych wejściowych jest nieodpowiedni z kilku powodów:

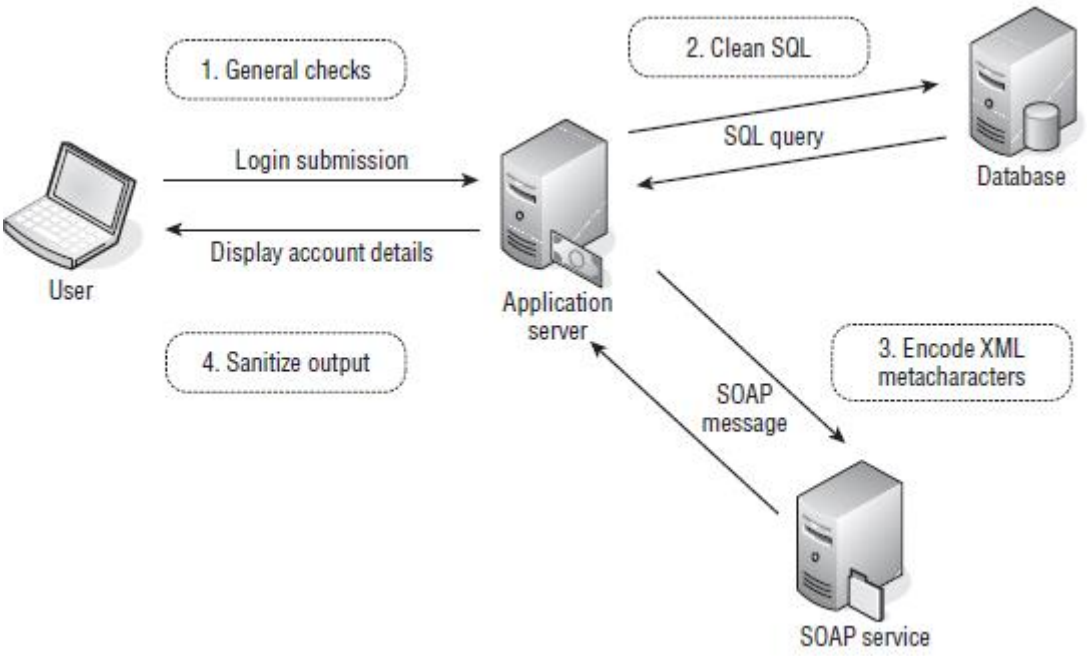
* Biorąc pod uwagę szeroki zakres funkcjonalności implementowanych przez aplikacje i różne stosowane technologie, typowa aplikacja musi bronić się przed ogromną różnorodnością ataków

opartych na danych wejściowych, z których każdy może wykorzystywać zróżnicowany zestaw spreparowanych danych. Bardzo trudno byłoby opracować jeden mechanizm na granicy zewnętrznej do obrony przed wszystkimi tymi atakami.

* Wiele funkcji aplikacji wymaga łączenia szeregu różnych rodzajów przetwarzania. Pojedynczy fragment danych wejściowych dostarczonych przez użytkownika może skutkować wieloma operacjami w różnych komponentach, przy czym dane wyjściowe każdego z nich są używane jako dane wejściowe dla następnego. W miarę przekształcania danych może się okazać, że nie będą one przypominać oryginalnych danych wejściowych. Wykwalifikowany atakujący może być w stanie manipulować aplikacją, aby spowodować wygenerowanie złośliwych danych wejściowych na kluczowym etapie przetwarzania, atakując komponent, który otrzymuje te dane. Niezwykle trudno byłoby wdrożyć mechanizm walidacji na granicy zewnętrznej, aby przewidzieć wszystkie możliwe wyniki przetwarzania każdego elementu danych wprowadzonych przez użytkownika.

* Obrona przed różnymi kategoriami ataków opartych na danych wejściowych może wiązać się z przeprowadzaniem różnych kontroli poprawności danych wprowadzanych przez użytkownika, które są ze sobą niezgodne. Na przykład zapobieganie atakom typu cross-site scripting może wymagać od aplikacji kodowania HTML znaku > jako >, a zapobieganie atakom polegającym na wstrzykiwaniu poleceń może wymagać od aplikacji blokowania danych wejściowych zawierających znaki & i ; postacie. Próba jednoczesnego zapobieżenia wszystkim kategoriom ataków na zewnętrznej granicy aplikacji może być czasem niemożliwa.

Bardziej efektywny model wykorzystuje koncepcję sprawdzania poprawności granic. Tutaj każdy komponent lub jednostka funkcjonalna aplikacji po stronie serwera traktuje swoje dane wejściowe jako pochodzące z potencjalnie złośliwego źródła. Sprawdzanie poprawności danych odbywa się na każdej z tych granic zaufania, oprócz zewnętrznej granicy między klientem a serwerem. Ten model zapewnia rozwiązanie opisanych problemów. Każdy komponent może się bronić przed określonymi typami spreparowanych danych wejściowych, na które może być podatny. Gdy dane przechodzą przez różne komponenty, można przeprowadzać kontrole sprawdzające poprawność względem dowolnej wartości, jaką dane mają w wyniku poprzednich przekształceń. A ponieważ różne kontrole walidacyjne są wdrażane na różnych etapach przetwarzania, jest mało prawdopodobne aby wchodziły ze sobą w konflikt. Rysunek ilustruje typową sytuację, w której sprawdzanie poprawności granic jest najskuteczniejszym podejściem do obrony przed złośliwymi danymi wejściowymi.



Logowanie użytkownika skutkuje wykonaniem kilku kroków przetwarzania danych wprowadzonych przez użytkownika, i odpowiednia walidacja jest przeprowadzana na każdym kroku:

1. Aplikacja otrzymuje dane logowania użytkownika. Procedura obsługi formularza sprawdza, czy każdy element wejściowy zawiera tylko dozwolone znaki, mieści się w określonym limicie długości i nie zawiera żadnych znanych sygnatur ataków.

2. Aplikacja wykonuje zapytanie SQL w celu zweryfikowania poświadczeń użytkownika. Aby zapobiec atakom typu SQL injection, wszystkie znaki wprowadzone przez użytkownika, które mogą zostać użyte do ataku na bazę danych, są zmieniane przed skonstruowaniem zapytania.

3. Jeśli logowanie się powiedzie, aplikacja przekazuje określone dane z profilu użytkownika do usługi SOAP w celu pobrania dalszych informacji o jej koncie. Aby zapobiec atakom typu SOAP injection, wszelkie metaznaki XML w danych profilu użytkownika są odpowiednio kodowane.

4. Aplikacja wyświetla informacje o koncie użytkownika z powrotem w przeglądarce użytkownika. Aby zapobiec atakom typu cross-site scripting, aplikacja koduje HTML wszelkie dane dostarczone przez użytkownika, które są osadzane na zwracanej stronie.

Konkretne luki w zabezpieczeniach i zabezpieczenia występujące w tym scenariuszu zostaną szczegółowo zbadane w dalszych rozdziałach. Gdyby odmiany tej funkcjonalności obejmowały przekazywanie danych do dalszych komponentów aplikacji, podobne zabezpieczenia musiałyby zostać zaimplementowane na odpowiednich granicach zaufania. Na przykład, jeśli nieudane logowanie spowodowało wysłanie przez aplikację wiadomości e-mail z ostrzeżeniem do użytkownika, wszelkie dane użytkownika zawarte w wiadomości e-mail mogą wymagać sprawdzenia pod kątem ataków iniekcji SMTP.

Walidacja wieloetapowa i kanonizacja

Częsty problem napotykany przez mechanizmy obsługi danych wejściowych pojawia się, gdy dane wejściowe dostarczone przez użytkownika są przetwarzane na kilku etapach w ramach logiki sprawdzania poprawności. Jeśli ten proces nie zostanie przeprowadzony ostrożnie, osoba atakująca może być w stanie skonstruować spreparowane dane wejściowe, które z powodzeniem przemycą

złośliwe dane przez mechanizm sprawdzania poprawności. Jedna wersja tego problemu występuje, gdy aplikacja próbuje oczyścić dane wprowadzane przez użytkownika przez usunięcie lub zakodowanie pewnych znaków lub wyrażeń. Na przykład aplikacja może próbować bronić się przed niektórymi atakami typu cross-site scripting, usuwając wyrażenie:

<script>

z jakichkolwiek danych dostarczonych przez użytkownika. Jednak osoba atakująca może być w stanie ominąć filtr, podając następujące dane wejściowe:

<scr<script>ipt>

Gdy zablokowane wyrażenie jest usuwane, otaczające dane kurczą się, aby przywrócić szkodliwy ładunek, ponieważ filtr nie jest stosowany rekurencyjnie. Podobnie, jeśli na danych wprowadzonych przez użytkownika zostanie przeprowadzony więcej niż jeden etap sprawdzania poprawności, osoba atakująca może wykorzystać kolejność tych kroków, aby ominąć filtr. Na przykład, jeśli aplikacja najpierw usunie ../ rekurencyjnie, a następnie rekurencyjnie usunie ..\, następujące dane wejściowe mogą zostać użyte do obejścia sprawdzania poprawności:

....\/

Podobny problem pojawia się w związku z kanonizacją danych. Gdy dane wejściowe są wysyłane z przeglądarki użytkownika, mogą być kodowane na różne sposoby. Te schematy kodowania istnieją po to, aby nietypowe znaki i dane binarne mogły być bezpiecznie przesyłane przez HTTP. Kanonizacja to proces konwersji lub dekodowania danych do wspólnego zestawu znaków. Jeśli jakakolwiek kanonizacja zostanie przeprowadzona po zastosowaniu filtrów wejściowych, osoba atakująca może być w stanie użyć odpowiedniego schematu kodowania, aby ominąć mechanizm sprawdzania poprawności. Na przykład aplikacja może próbować bronić się przed niektórymi atakami typu SQL injection, blokując dane wejściowe zawierające znak apostrofu. Jeśli jednak dane wejściowe zostaną następnie kanonizowane, osoba atakująca może być w stanie użyć podwójnego kodowania adresu URL, aby pokonać filtr. Na przykład:

%2527

Po odebraniu tych danych wejściowych serwer aplikacji przeprowadza normalne dekodowanie adresu URL, więc dane wejściowe mają postać:

%27

To nie zawiera apostrofu, więc jest dozwolone przez filtry aplikacji. Ale kiedy aplikacja wykonuje dalsze dekodowanie adresu URL, dane wejściowe są konwertowane na apostrof, co pozwala ominąć filtr. Jeśli aplikacja usuwa apostrof zamiast go blokować, a następnie przeprowadza dalszą kanonizację, skuteczne może być następujące obejście:

%%2727

Warto zauważyć, że wiele etapów walidacji i kanonizacji w tych przypadkach nie musi odbywać się po stronie serwera aplikacji. Na przykład w poniższym wejściu kilka znaków zostało zakodowanych w formacie HTML:

<iframe src=javasc&#x72ipt:alert(1) >

Jeśli aplikacja po stronie serwera używa filtra wejściowego do blokowania pewnych wyrażeń i znaków JavaScript, zakodowane dane wejściowe mogą pomyślnie ominąć filtr. Jeśli jednak dane wejściowe

zostaną następnie skopiowane do odpowiedzi aplikacji, niektóre przeglądarki wykonują dekodowanie HTML wartości parametru src i wykonywany jest osadzony JavaScript. Oprócz standardowych schematów kodowania, które są przeznaczone do użytku w aplikacjach internetowych, problemy z kanonizacją mogą pojawić się w innych sytuacjach, gdy komponent używany przez aplikację konwertuje dane z jednego zestawu znaków na inny. Na przykład niektóre technologie wykonują „najlepiej dopasowane" mapowanie znaków w oparciu o podobieństwa w drukowanych glifach. W tym przypadku znaki « i » mogą zostać przekształcone odpowiednio na < i >, a Ÿ i Â na Y i A. To zachowanie często można wykorzystać do przemycenia zablokowanych znaków lub słów kluczowych przez filtry wejściowe aplikacji. Opiszemy wiele tego rodzaju ataków, które są skuteczne w pokonywaniu mechanizmów obronnych wielu aplikacji przed typowymi lukami opartymi na danych wejściowych. Unikanie problemów z wieloetapową walidacją i kanonizacją może być czasami trudne i nie ma jednego rozwiązania problemu. Jednym ze sposobów jest rekurencyjne wykonywanie kroków sanityzacji, aż do momentu, gdy nie zostaną wprowadzone żadne dalsze modyfikacje elementu wejściowego. Jednakże, gdy pożądane sanityzacja obejmuje ucieczkę od problematycznego znaku, może to skutkować nieskończoną pętlą. Często problem można rozwiązać tylko indywidualnie, w oparciu o rodzaje przeprowadzanej walidacji. Tam, gdzie jest to wykonalne, lepiej jest unikać prób usunięcia niektórych rodzajów złych danych wejściowych i po prostu całkowicie je odrzucić.

Postępowanie z napastnikami

Każdy, kto projektuje aplikację, dla której bezpieczeństwo jest ważne, musi założyć, że będzie ona bezpośrednio atakowana przez wyspecjalizowanych i wykwalifikowanych atakujących. Kluczową funkcją mechanizmów bezpieczeństwa aplikacji jest możliwość obsługi i reagowania na te ataki w sposób kontrolowany. Mechanizmy te często obejmują kombinację środków obronnych i ofensywnych, których celem jest jak największa frustracja atakującego oraz zapewnienie właścicielom aplikacji odpowiednich powiadomień i dowodów na to, co się wydarzyło. Środki wdrożone w celu obsługi atakujących zazwyczaj obejmują następujące zadania:

* Obsługa błędów

* Prowadzenie dzienników audytów

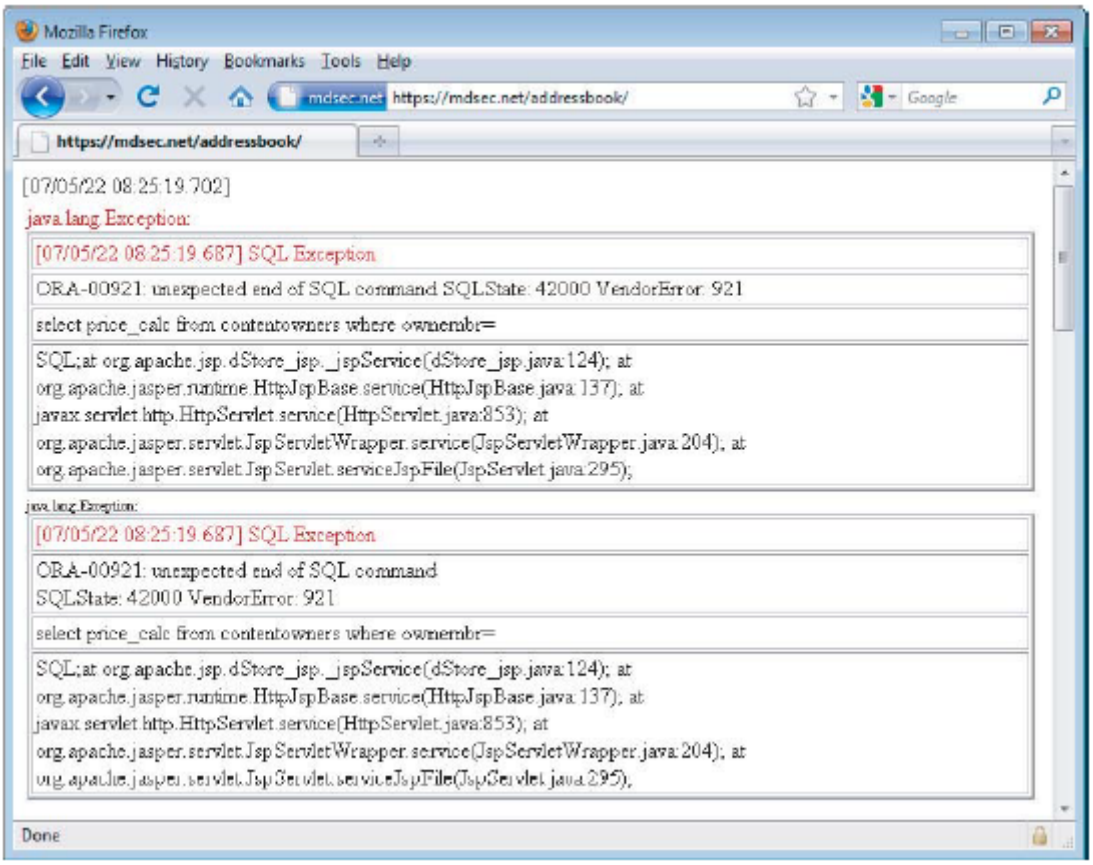
* Alarmowanie administratorów

* Reagowanie na ataki

Obsługa błędów

Niezależnie od tego, jak ostrożni są programiści aplikacji podczas sprawdzania poprawności danych wejściowych sera, jest praktycznie nieuniknione, że wystąpią pewne nieprzewidziane błędy. Błędy wynikające z działań zwykłych użytkowników zostaną prawdopodobnie zidentyfikowane podczas testów funkcjonalności i akceptacji użytkowników. W związku z tym są one brane pod uwagę przed wdrożeniem aplikacji w kontekście produkcyjnym. Trudno jednak przewidzieć każdy możliwy sposób interakcji złośliwego użytkownika z aplikacją, dlatego należy spodziewać się dalszych błędów, gdy aplikacja zostanie zaatakowana. Kluczowym mechanizmem obronnym jest, aby aplikacja z wdziękiem radziła sobie z nieoczekiwanymi błędami i albo naprawiała je, albo wyświetlała użytkownikowi odpowiedni komunikat o błędzie. W kontekście produkcyjnym aplikacja nigdy nie powinna zwracać żadnych komunikatów generowanych przez system ani innych informacji debugowania w swoich odpowiedziach. Jak zobaczysz w tej książce, zbyt szczegółowe komunikaty o błędach mogą znacznie pomóc szkodliwym użytkownikom w przeprowadzaniu ataków na aplikację. W niektórych sytuacjach osoba atakująca może wykorzystać wadliwą obsługę błędów do odzyskania poufnych informacji

zawartych w samych komunikatach o błędach, zapewniając cenny kanał do kradzieży danych z aplikacji. Rysunek przedstawia przykład nieobsługiwanego błędu skutkującego wyświetleniem szczegółowego komunikatu o błędzie.



Większość języków tworzenia stron internetowych zapewnia dobrą obsługę błędów poprzez bloki try-catch i sprawdzone wyjątki. Kod aplikacji powinien szeroko wykorzystywać te konstrukcje, aby wyłapywać określone i ogólne błędy oraz odpowiednio je obsługiwać. Co więcej, większość serwerów aplikacji można skonfigurować tak, aby radziły sobie z nieobsługiwanymi błędami aplikacji w niestandardowy sposób, na przykład wyświetlając nieinformacyjny komunikat o błędzie.

Skuteczna obsługa błędów jest często zintegrowana z mechanizmami logowania aplikacji, które rejestrują jak najwięcej informacji debugowania o nieprzewidzianych błędach. Nieoczekiwane błędy często wskazują na defekty w zabezpieczeniach aplikacji, które można usunąć u źródła, jeśli właściciel aplikacji posiada wymagane informacje.

Prowadzenie dzienników audytów

Dzienniki kontroli są przydatne przede wszystkim podczas badania prób włamań do aplikacji. Skuteczne logi audytu po takim incydencie powinny umożliwiać właścicielom aplikacji dokładne zrozumienie, co się wydarzyło, jakie luki (jeśli w ogóle) zostały wykorzystane, czy atakujący uzyskał nieautoryzowany dostęp do danych lub wykonał nieautoryzowane działania oraz w miarę możliwości , dostarczyć dowód tożsamości intruza. W każdej aplikacji, dla której bezpieczeństwo jest ważne, kluczowe zdarzenia powinny być rejestrowane jako rzecz oczywista. Jako minimum zazwyczaj obejmują one:

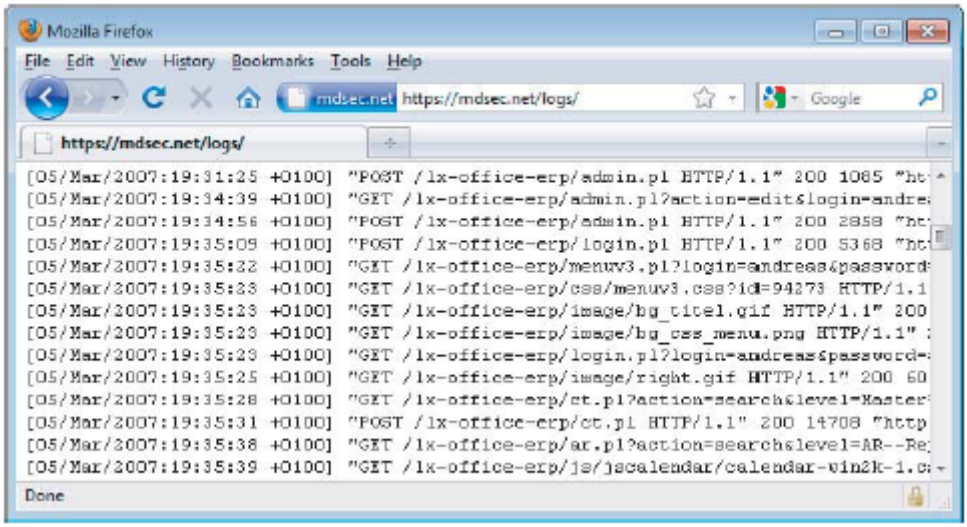
* Wszystkie zdarzenia związane z funkcjonalnością uwierzytelniania, takie jak udane i nieudane logowanie oraz zmiana hasła

* Kluczowe transakcje, takie jak płatności kartą kredytową i transfery środków

* Próby dostępu, które są blokowane przez mechanizmy kontroli dostępu

* Wszelkie żądania zawierające znane ciągi ataku, które wskazują na jawnie złośliwe zamiary

W wielu aplikacjach o krytycznym znaczeniu dla bezpieczeństwa, takich jak te używane przez banki internetowe, każde żądanie klienta jest w pełni rejestrowane, co zapewnia pełny zapis dochodzeniowy, który można wykorzystać do zbadania wszelkich incydentów. Skuteczne dzienniki audytu zazwyczaj rejestrują czas każdego zdarzenia, adres IP, z którego otrzymano żądanie, oraz konto użytkownika (jeśli zostało uwierzytelnione). Takie dzienniki muszą być silnie chronione przed nieautoryzowanym dostępem do odczytu lub zapisu. Skutecznym podejściem jest przechowywanie dzienników audytu w autonomicznym systemie, który akceptuje tylko komunikaty aktualizacyjne z głównej aplikacji. W niektórych sytuacjach dzienniki mogą zostać przeniesione na nośniki jednokrotnego zapisu, aby zapewnić ich integralność w przypadku udanego ataku. Jeśli chodzi o powierzchnię ataku, słabo chronione dzienniki audytu mogą dostarczyć atakującemu kopalnię złota, ujawniając wiele poufnych informacji, takich jak tokeny sesji i parametry żądań. Te informacje mogą umożliwić atakującemu natychmiastowe złamanie zabezpieczeń całej aplikacji, jak pokazano na rysunku



Alert dla administratorów

Dzienniki audytu umożliwiają właścicielom aplikacji retrospektywne badanie prób włamań i, jeśli to możliwe, podjęcie kroków prawnych przeciwko sprawcy. Jednak w wielu sytuacjach pożądane jest podejmowanie znacznie bardziej natychmiastowych działań, w czasie rzeczywistym, w odpowiedzi na próby ataków. Na przykład administratorzy mogą zablokować adres IP lub konto użytkownika używane przez osobę atakującą. W skrajnych przypadkach mogą nawet wyłączyć aplikację na czas badania ataku i podjęcia działań zaradczych. Nawet jeśli udane wtargnięcie już miało miejsce, jego praktyczne skutki mogą zostać złagodzone, jeśli działania obronne zostaną podjęte na wczesnym etapie. W większości sytuacji mechanizmy ostrzegania muszą równoważyć sprzeczne cele polegające na rzetelnym zgłaszaniu każdego prawdziwego ataku i unikaniu generowania tylu alertów, które mogłyby zostać zignorowane. Dobrze zaprojektowany mechanizm alertów może wykorzystywać kombinację czynników do diagnozowania, że trwa określony atak, i w miarę możliwości może agregować powiązane zdarzenia w jeden alert. Anomalne zdarzenia monitorowane przez mechanizmy alertów często obejmują:

* Anomalie użytkowania, takie jak duża liczba żądań otrzymywanych z pojedynczego adresu IP lub użytkownika, wskazujące na atak skryptowy

* Anomalie biznesowe, takie jak nietypowa liczba przelewów środków na lub z jednego konta bankowego

* Żądania zawierające znane ciągi ataków

* Żądania, w których zmodyfikowano dane ukryte przed zwykłymi użytkownikami

Niektóre z tych funkcji mogą być dość dobrze zapewniane przez gotowe zapory aplikacyjne i produkty do wykrywania włamań. Zazwyczaj używają one kombinacji reguł opartych na sygnaturach i anomaliach w celu identyfikacji złośliwego użycia aplikacji i mogą reaktywnie blokować złośliwe żądania, a także wysyłać alerty do administratorów. Produkty te mogą stanowić cenną warstwę ochronną chroniącą aplikację internetową, szczególnie w przypadku istniejących aplikacji, o których wiadomo, że zawierają problemy, ale w przypadku których zasoby umożliwiające ich rozwiązanie nie są natychmiast dostępne. Jednak ich skuteczność jest zwykle ograniczona przez fakt, że każda aplikacja internetowa jest inna, więc stosowane reguły są nieuchronnie do pewnego stopnia ogólne. Zapory sieciowe zazwyczaj dobrze identyfikują najbardziej oczywiste ataki, w przypadku których osoba atakująca przesyła standardowe ciągi ataku w każdym parametrze żądania. Jednak wiele ataków jest bardziej subtelnych. Na przykład, być może modyfikują numer konta w ukrytym polu, aby uzyskać dostęp do danych innego użytkownika, lub wysyłają żądania poza kolejnością, aby wykorzystać defekty w logice aplikacji. W takich przypadkach żądanie przesłane przez osobę atakującą może być identyczne z żądaniem przesłanym przez nieszkodliwego użytkownika. To, co czyni go złośliwym, to okoliczności, w których jest tworzony. W każdej aplikacji o krytycznym znaczeniu dla bezpieczeństwa najskuteczniejszym sposobem wdrożenia alertów w czasie rzeczywistym jest ścisła integracja z mechanizmami sprawdzania poprawności danych wejściowych aplikacji i innymi kontrolami. Na przykład, jeśli plik cookie ma mieć jeden z określonych zestawów wartości, każde naruszenie tego oznacza, że jego wartość została zmodyfikowana w sposób, który nie jest możliwy dla zwykłych użytkowników aplikacji. Podobnie, jeśli użytkownik zmieni numer konta w ukrytym polu, aby zidentyfikować konto innego użytkownika, zdecydowanie wskazuje to na złośliwe zamiary. Aplikacja powinna już sprawdzać te ataki w ramach swojej podstawowej obrony, a te mechanizmy ochronne można łatwo podłączyć do mechanizmu alertów aplikacji, aby zapewnić w pełni dostosowane wskaźniki złośliwej aktywności. Ponieważ kontrole te zostały dostosowane do rzeczywistej logiki aplikacji, z precyzyjną wiedzą o tym, jak powinni zachowywać się zwykli użytkownicy, są one znacznie mniej podatne na fałszywe alarmy niż jakiekolwiek gotowe rozwiązanie, niezależnie od tego, jak konfigurowalne lub łatwe w użyciu dowiedzieć się, że rozwiązanie może być.

Reagowanie na ataki

Oprócz ostrzegania administratorów wiele aplikacji o znaczeniu krytycznym dla bezpieczeństwa zawiera wbudowane mechanizmy defensywnej reakcji na użytkowników, którzy zostali zidentyfikowani jako potencjalnie złośliwi. Ponieważ każda aplikacja jest inna, większość ataków w świecie rzeczywistym wymaga od osoby atakującej systematycznego sondowania luk w zabezpieczeniach, wysyłania wielu żądań zawierających spreparowane dane wejściowe mające na celu wskazanie obecności różnych typowych luk w zabezpieczeniach. Skuteczne mechanizmy sprawdzania poprawności danych wejściowych identyfikują wiele z tych żądań jako potencjalnie złośliwe i blokują dane wejściowe przed niepożądanym wpływem na aplikację. Rozsądnie jest jednak założyć, że istnieją pewne obejścia tych filtrów i że aplikacja zawiera rzeczywiste luki, które czekają na wykrycie i wykorzystanie. W pewnym momencie atakujący pracujący systematycznie może odkryć te wady. Z tego powodu niektóre aplikacje podejmują automatyczne działania reaktywne, aby udaremnić

działania atakującego, który działa w ten sposób. Mogą na przykład coraz wolniej odpowiadać na żądania atakującego lub zakończyć sesję atakującego, wymagając od niego zalogowania się lub wykonania innych czynności przed kontynuowaniem ataku. Chociaż te środki nie pokonają najbardziej cierpliwego i zdeterminowanego atakującego, odstraszą wielu przypadkowych napastników i zapewnią administratorom dodatkowy czas na monitorowanie sytuacji i podjęcie bardziej drastycznych działań, jeśli zajdzie taka potrzeba. Reagowanie na pozornych napastników nie zastępuje oczywiście naprawiania istniejących luk w aplikacji. Jednak w prawdziwym świecie nawet najbardziej skrupulatne wysiłki mające na celu usunięcie luk w zabezpieczeniach aplikacji mogą pozostawić pewne defekty, które można wykorzystać. Umieszczanie dalszych przeszkód na drodze atakującego jest skutecznym środkiem dogłębnej obrony, który zmniejsza prawdopodobieństwo wykrycia i wykorzystania wszelkich pozostałych luk w zabezpieczeniach.

Zarządzanie aplikacją

Każda użyteczna aplikacja musi być zarządzana i administrowana. Ta funkcja często stanowi kluczową część mechanizmów bezpieczeństwa aplikacji, umożliwiając administratorom zarządzanie kontami i rolami użytkowników, dostęp do funkcji monitorowania i audytu, wykonywanie zadań diagnostycznych i konfigurowanie aspektów funkcjonalności aplikacji. W wielu aplikacjach funkcje administracyjne są zaimplementowane w samej aplikacji, dostępnej za pośrednictwem tego samego interfejsu WWW, co jej podstawowe funkcje niezwiązane z bezpieczeństwem, jak pokazano na rysunku 2.8. W takim przypadku mechanizm administracyjny stanowi krytyczną część powierzchni ataku aplikacji. Jego główną atrakcją dla atakującego jest możliwość eskalacji uprawnień. Na przykład:

* Luki w mechanizmie uwierzytelniania mogą umożliwić atakującemu uzyskanie dostępu administracyjnego, skutecznie zagrażając całej aplikacji.

* Wiele aplikacji nie wdraża skutecznej kontroli dostępu do niektórych swoich funkcji administracyjnych. Osoba atakująca może znaleźć sposób na utworzenie nowego konta użytkownika z potężnymi uprawnieniami.

* Funkcjonalność administracyjna często polega na wyświetlaniu danych pochodzących od zwykłych użytkowników. Wszelkie luki związane ze skryptami krzyżowymi w interfejsie administracyjnym mogą prowadzić do naruszenia bezpieczeństwa sesji użytkownika, która ma zagwarantowane potężne uprawnienia.

* Funkcjonalności administracyjne są często poddawane mniej rygorystycznym testom bezpieczeństwa, ponieważ ich użytkownicy są uważani za zaufanych lub ponieważ osoby przeprowadzające testy penetracyjne mają dostęp tylko do kont o niskich uprawnieniach. Ponadto funkcjonalność ta często wymaga wykonywania z natury niebezpiecznych operacji, obejmujących dostęp do plików na dysku lub polecenia systemu operacyjnego. Jeśli osoba atakująca może zagrozić funkcji administracyjnej, często może ją wykorzystać do przejęcia kontroli nad całym serwerem.

Streszczenie

Pomimo znacznych różnic, praktycznie wszystkie aplikacje internetowe wykorzystują te same podstawowe mechanizmy bezpieczeństwa w jakimś kształcie lub formie. Mechanizmy te stanowią podstawową obronę aplikacji przed złośliwymi użytkownikami, a zatem stanowią również większość obszaru ataku aplikacji. Luki, które przyjrzymy się później, wynikają głównie z defektów w tych podstawowych mechanizmach. Spośród tych składników najważniejsze są mechanizmy obsługi dostępu użytkownika i danych wprowadzanych przez użytkownika, na które należy zwrócić największą uwagę podczas kierowania aplikacji. Wady tych mechanizmów często prowadzą do całkowitego

skompromitowania aplikacji, umożliwiając dostęp do danych należących do innych użytkowników, wykonywanie nieautoryzowanych działań oraz wstrzykiwanie dowolnego kodu i poleceń.