Atakowanie logiki aplikacji

Wszystkie aplikacje internetowe wykorzystują logikę do dostarczania swojej funkcjonalności. Pisanie kodu w języku programowania polega u podstaw na niczym innym, jak rozbiciu złożonego procesu na proste i dyskretne logiczne kroki. Przełożenie funkcji, która ma znaczenie dla ludzi, na sekwencję małych operacji, które może wykonać komputer, wymaga dużej zręczności i dyskrecji. Robienie tego w elegancki i bezpieczny sposób jest jeszcze trudniejsze. Gdy duża liczba różnych projektantów i programistów pracuje równolegle nad tą samą aplikacją, istnieje duże prawdopodobieństwo wystąpienia błędów. We wszystkich aplikacjach internetowych, z wyjątkiem tych najprostszych, na każdym etapie wykonywana jest ogromna ilość logiki. Ta logika przedstawia skomplikowaną powierzchnię ataku, która jest zawsze obecna, ale często pomijana. Wiele przeglądów kodu i testów penetracyjnych koncentruje się wyłącznie na typowych „głównych" lukach w zabezpieczeniach, takich jak iniekcja SQL i skrypty między witrynami, ponieważ mają one łatwo rozpoznawalną sygnaturę i dobrze zbadany wektor wykorzystania. Z drugiej strony, luki w logice aplikacji są trudniejsze do scharakteryzowania: każda instancja może wydawać się wyjątkowym, jednorazowym zdarzeniem i zazwyczaj nie są one identyfikowane przez żadne automatyczne skanery podatności. W rezultacie na ogół nie są one tak dobrze doceniane ani rozumiane, a zatem są bardzo interesujące dla osoby atakującej. W tym rozdziale opisano rodzaje błędów logicznych, które często występują w aplikacjach internetowych, oraz praktyczne kroki, które można podjąć, aby zbadać i zaatakować logikę aplikacji. Przedstawimy serię rzeczywistych przykładów, z których każdy przejawia inny rodzaj defektu logicznego. Razem ilustrują one różnorodność założeń przyjmowanych przez projektantów i programistów, które mogą prowadzić bezpośrednio do błędnej logiki i narażać aplikację na luki w zabezpieczeniach.

Natura błędów logicznych

Błędy logiczne w aplikacjach internetowych są niezwykle zróżnicowane. Obejmują one zarówno proste błędy przejawiające się w kilku liniach kodu, jak i złożone luki wynikające ze współpracy kilku podstawowych komponentów aplikacji. W niektórych przypadkach mogą być oczywiste i łatwe do wykrycia; w innych przypadkach mogą być wyjątkowo subtelne i mogą umknąć nawet najbardziej rygorystycznej ocenie kodu lub testowi penetracyjnemu. W przeciwieństwie do innych błędów kodowania, takich jak wstrzykiwanie kodu SQL lub skrypty między witrynami, żadna wspólna „sygnatura" nie jest powiązana z błędami logicznymi. Charakterystyczną cechą jest oczywiście to, że logika zaimplementowana w aplikacji jest w jakiś sposób wadliwa. W wielu przypadkach defekt można przedstawić w postaci konkretnego założenia przyjętego przez projektanta lub programistę, jawnie lub niejawnie, które okazuje się błędne. Ogólnie rzecz biorąc, programista mógł rozumować na przykład: „Jeśli wydarzy się A, to B musi się wydarzyć, więc zrobię C". Programista nie zadał zupełnie innego pytania „Ale co, jeśli wystąpi X?" i dlatego + nie wziął pod uwagę scenariusza, który narusza to założenie. W zależności od okoliczności to błędne założenie może spowodować powstanie znacznej luki w zabezpieczeniach. Ponieważ w ostatnich latach wzrosła świadomość powszechnych luk w zabezpieczeniach aplikacji internetowych, częstość występowania i dotkliwość niektórych kategorii luk znacznie spadła. Jednak ze względu na charakter błędów logicznych jest mało prawdopodobne, że zostaną one kiedykolwiek wyeliminowane za pomocą standardów bezpiecznego programowania, użycia narzędzi do audytu kodu lub normalnych testów penetracyjnych. Zróżnicowany charakter błędów logicznych oraz fakt, że wykrywanie ich i zapobieganie im często wymaga dużej dozy myślenia lateralnego sugeruje, że będą one powszechne jeszcze przez długi czas. Dlatego każdy poważny atakujący musi zwrócić szczególną uwagę na logikę stosowaną w aplikacji, która jest celem ataku, aby spróbować zrozumieć założenia, które prawdopodobnie przyjęli projektanci i programiści. Następnie powinien pomyśleć z wyobraźnią o tym, jak te założenia mogą zostać naruszone.

Błędy logiczne w świecie rzeczywistym

Najlepszym sposobem poznania błędów logicznych nie jest teoretyzowanie, ale zapoznanie się z konkretnymi przykładami. Chociaż poszczególne przypadki błędów logicznych znacznie się różnią, mają wiele wspólnych motywów i pokazują rodzaje błędów, które programiści zawsze będą skłonni popełniać. Dlatego spostrzeżenia zebrane podczas studiowania próbki błędów logicznych powinny pomóc ci odkryć nowe błędy w zupełnie innych sytuacjach.

Przykład 1: Pytanie do Wyroczni

Autorzy znaleźli przypadki luki „szyfrującej wyroczni" w wielu różnych typach aplikacji. Używali go w wielu atakach, od odszyfrowywania poświadczeń domeny w oprogramowaniu do drukowania po łamanie przetwarzania w chmurze. Poniżej znajduje się klasyczny przykład usterki znalezionej w witrynie sprzedaży oprogramowania.

Funkcjonalność

Aplikacja zaimplementowała funkcję „zapamiętaj mnie", dzięki której użytkownik mógł uniknąć logowania do aplikacji przy każdej wizycie, pozwalając aplikacji na ustawienie stałego pliku cookie w przeglądarce. Ten plik cookie był chroniony przed manipulacją lub ujawnieniem przez algorytm szyfrowania, który był uruchamiany na ciągu znaków składającym się z nazwy, identyfikatora użytkownika i nietrwałych danych, aby zapewnić, że wynikowa wartość jest unikalna i nie można jej przewidzieć. Aby uniemożliwić jej odtworzenie przez atakującego, który uzyskał do niej dostęp, zbierano również dane specyficzne dla maszyny, w tym adres IP. Ten plik cookie został słusznie uznany za solidne rozwiązanie do ochrony potencjalnie podatnej na ataki części wymaganej funkcjonalności biznesowej.

Oprócz funkcji „zapamiętaj mnie", aplikacja miała funkcję przechowywania nazwy ekranowej użytkownika w pliku cookie o nazwie ScreenName. W ten sposób użytkownik mógł otrzymać spersonalizowane powitanie w rogu strony przy każdej następnej wizycie na stronie. Decydując, że ta nazwa była również częścią informacji bezpieczeństwa, uznano, że należy ją również zaszyfrować.

Założenie

Twórcy zdecydowali, że ponieważ plik cookie ScreenName ma znacznie mniejszą wartość dla atakującego niż plik cookie RememberMe, równie dobrze mogą użyć tego samego algorytmu szyfrowania do jego ochrony. Nie wzięli pod uwagę tego, że użytkownik może podać swoją nazwę ekranową i wyświetlić ją na ekranie. To nieumyślnie dało użytkownikom dostęp do funkcji szyfrowania (i klucza szyfrowania) używanej do ochrony trwałego tokena uwierzytelniania RememberMe.

Atak

W prostym ataku użytkownik podał zaszyfrowaną wartość swojego pliku cookie RememberMe zamiast zaszyfrowanego pliku cookie ScreenName. Podczas wyświetlania użytkownikowi nazwy ekranowej aplikacja odszyfrowuje wartość, sprawdza, czy odszyfrowanie zadziałało, a następnie wyświetla wynik na ekranie. Spowodowało to następujący komunikat:

Witaj Marcusie|734|192.168.4.282750184

Chociaż było to interesujące, niekoniecznie stanowiło problem wysokiego ryzyka. Oznaczało to po prostu, że atakujący, mając zaszyfrowany plik cookie RememberMe, mógł wyświetlić zawartość, w tym nazwę użytkownika, identyfikator użytkownika i adres IP. Ponieważ w pliku cookie nie było przechowywane żadne hasło, nie było możliwości natychmiastowego działania na podstawie

uzyskanych informacji. Prawdziwy problem wynikał z faktu, że użytkownicy mogli określić swoje nazwy ekranowe. W rezultacie użytkownik może wybrać tę nazwę ekranową, na przykład:

admin|1|192.168.4.282750184

Gdy użytkownik wylogował się i zalogował ponownie, aplikacja zaszyfrowała tę wartość i zapisała ją w przeglądarce użytkownika jako zaszyfrowany plik cookie ScreenName. Jeśli atakujący przesłał ten zaszyfrowany token jako wartość pliku cookie RememberMe, aplikacja odszyfrowała go, odczytała identyfikator użytkownika i zalogowała atakującego jako administratora! Mimo że szyfrowanie było potrójne DES, przy użyciu silnego klucza i chronione przed atakami powtórkowymi, aplikacja mogła zostać wykorzystana jako „wyrocznia szyfrowania" do odszyfrowywania i szyfrowania dowolnych wartości.

KROKI HACKOWANIA

Manifestacje tego typu luk można znaleźć w różnych lokalizacjach. Przykłady obejmują tokeny odzyskiwania konta, oparty na tokenach dostęp do uwierzytelnionych zasobów i wszelkie inne wartości wysyłane po stronie klienta, które muszą być odporne na manipulacje lub nieczytelne dla użytkownika.

1. Poszukaj lokalizacji, w których w aplikacji stosowane jest szyfrowanie (nie haszowanie). Określ wszystkie lokalizacje, w których aplikacja szyfruje lub odszyfrowuje wartości dostarczone przez użytkownika, i spróbuj zastąpić wszelkie inne zaszyfrowane wartości napotkane w aplikacji. Spróbuj spowodować błąd w aplikacji, który ujawnia odszyfrowaną wartość lub gdzie odszyfrowana wartość jest celowo wyświetlana na ekranie.

2. Poszukaj luki w zabezpieczeniach „ujawniania wyroczni", określając, gdzie można podać zaszyfrowaną wartość, która spowoduje wyświetlenie odpowiedniej odszyfrowanej wartości w odpowiedzi aplikacji. Ustal, czy prowadzi to do ujawnienia poufnych informacji, takich jak hasło lub karta kredytowa.

3. Poszukaj luki w zabezpieczeniach „oracle encrypt", określając, gdzie podanie wartości w postaci zwykłego tekstu powoduje, że aplikacja zwraca odpowiednią zaszyfrowaną wartość. Określ, gdzie może to zostać nadużyte, określając dowolne wartości lub złośliwe ładunki, które aplikacja będzie przetwarzać.

Przykład 2: Oszukanie funkcji zmiany hasła

Autorzy napotkali ten błąd logiczny w aplikacji internetowej zaimplementowanej przez firmę świadczącą usługi finansowe, a także w aplikacji AOL AIM Enterprise Gateway.

Funkcjonalność

Aplikacja zaimplementowała funkcję zmiany hasła dla użytkowników końcowych. Wymagało to od użytkownika wypełnienia pól dotyczących nazwy użytkownika, istniejącego hasła, nowego hasła i potwierdzenia nowego hasła. Nie zabrakło również funkcji zmiany hasła do użytku przez administratorów. To pozwoliło im zmienić hasło dowolnego użytkownika bez podawania istniejącego hasła. Obie funkcje zostały zaimplementowane w tym samym skrypcie po stronie serwera.

Założenie

Interfejs po stronie klienta prezentowany użytkownikom i administratorom różnił się pod jednym względem: interfejs administratora nie zawierał pola na dotychczasowe hasło. Gdy aplikacja po stronie serwera przetwarzała żądanie zmiany hasła, wykorzystywała obecność lub brak istniejącego parametru hasła do wskazania, czy żądanie pochodzi od administratora, czy od zwykłego użytkownika. Innymi

słowy, założono, że zwykli użytkownicy zawsze będą podawać istniejący parametr hasła. Odpowiedzialny kod wyglądał mniej więcej tak:

```
String existingPassword = request.getParameter(“existingPassword”);

if (null == existingPassword)

{

trace(“Old password not supplied, must be an administrator”);

return true;

}

else

{

trace(“Verifying user’s old password”);

...
```

Atak

Kiedy założenie jest wyraźnie sformułowane w ten sposób, błąd logiczny staje się oczywisty. Oczywiście zwykły użytkownik mógłby wysłać żądanie, które nie zawierałoby istniejącego parametru hasła, ponieważ użytkownicy kontrolowali każdy aspekt wysyłanych przez siebie żądań. Ta wada logiczna była druzgocąca dla aplikacji. Umożliwiło to atakującemu zresetowanie hasła dowolnego innego użytkownika i przejęcie pełnej kontroli nad kontem tej osoby.

KROKI HACKOWANIA

1. Podczas sondowania kluczowych funkcji pod kątem błędów logicznych, spróbuj usunąć po kolei każdy parametr przesłany w żądaniach, w tym pliki cookie, pola ciągów zapytań i elementy danych POST.

2. Pamiętaj, aby usunąć rzeczywistą nazwę parametru oraz jego wartość. Nie przesyłaj po prostu pustego ciągu znaków, ponieważ zazwyczaj serwer obsługuje to inaczej.

3. Atakuj tylko jeden parametr na raz, aby zapewnić osiągnięcie wszystkich odpowiednich ścieżek kodu w aplikacji.

4. Jeśli żądanie, którym manipulujesz, jest częścią procesu wieloetapowego, śledź proces aż do zakończenia, ponieważ późniejsza logika może przetwarzać dane dostarczone we wcześniejszych krokach i przechowywane w ramach sesji.

Przykład 3: przejście do kasy

Autorzy napotkali tę lukę logiczną w aplikacji internetowej używanej przez sprzedawcę internetowego.

Funkcjonalność

Proces składania zamówienia składał się z następujących etapów:

1. Przejrzyj katalog produktów i dodaj pozycje do koszyka.

2. Wróć do koszyka i sfinalizuj zamówienie.

3. Wprowadź informacje o płatności.

4. Wprowadź informacje o dostawie.

Założenie

Twórcy założyli, że użytkownicy będą zawsze uzyskiwać dostęp do etapów w zamierzonej kolejności, ponieważ taka jest kolejność, w jakiej etapy są dostarczane użytkownikowi przez linki nawigacyjne i formularze prezentowane w przeglądarce użytkownika. W związku z tym każdy użytkownik, który zakończył proces zamawiania, musiał złożyć zadowalające szczegóły płatności po drodze.

Atak

Założenie twórców zostało sfałszowane z dość oczywistych powodów. Użytkownicy kontrolowali każde żądanie kierowane do aplikacji, dzięki czemu mogli uzyskać dostęp do dowolnego etapu procesu zamawiania w dowolnej kolejności. Przechodząc bezpośrednio z etapu 2 do etapu 4, osoba atakująca może wygenerować zamówienie, które zostało sfinalizowane do dostawy, ale w rzeczywistości nie zostało opłacone.

KROKI HACKOWANIA

Technika znajdowania i wykorzystywania tego rodzaju luk jest znana jako wymuszone przeglądanie. Polega na obejściu wszelkich kontroli nałożonych przez nawigację w przeglądarce na kolejność uzyskiwania dostępu do funkcji aplikacji:

1. Gdy proces wieloetapowy obejmuje zdefiniowaną sekwencję żądań, spróbuj przesłać te żądania poza oczekiwaną kolejnością. Spróbuj pominąć niektóre etapy, uzyskując dostęp do jednego etapu więcej niż raz i uzyskując dostęp do wcześniejszych etapów po późniejszych.

2. Dostęp do sekwencji etapów można uzyskać za pomocą serii żądań GET lub POST dla różnych adresów URL lub mogą one obejmować przesyłanie różnych zestawów parametrów do tego samego adresu URL. Żądany etap można określić, podając nazwę funkcji lub indeks w parametrze żądania. Upewnij się, że w pełni rozumiesz mechanizmy stosowane przez aplikację w celu zapewnienia dostępu do różnych etapów.

3. Z kontekstu zaimplementowanej funkcjonalności spróbuj zrozumieć, jakie założenia mogli przyjąć programiści i gdzie leży kluczowa powierzchnia ataku. Spróbuj zidentyfikować sposoby naruszenia tych założeń, aby spowodować niepożądane zachowanie w aplikacji.

4. Gdy dostęp do funkcji wieloetapowych odbywa się poza kolejnością, w aplikacji często występują różne anomalie, takie jak zmienne z wartościami pustymi lub niezainicjowanymi, stan częściowo zdefiniowany lub niespójny oraz inne nieprzewidywalne zachowania. W takiej sytuacji aplikacja może zwrócić interesujący komunikat o błędzie i wynik debugowania, których można użyć do lepszego zrozumienia jej wewnętrznego działania i tym samym dostrojenia bieżącego lub innego ataku Czasami aplikacja może wejść w stan całkowicie nieoczekiwany przez programistów , co może prowadzić do poważnych luk w zabezpieczeniach.

NOTATKA : Wiele rodzajów luk w zabezpieczeniach związanych z kontrolą dostępu ma podobny charakter do tej usterki logicznej. Gdy funkcja uprzywilejowana obejmuje wiele etapów, które normalnie są dostępne w określonej kolejności, aplikacja może założyć, że użytkownicy zawsze będą przechodzić przez tę funkcjonalność w tej kolejności. Aplikacja może wymusić ścisłą kontrolę dostępu na początkowych etapach procesu i zakładać, że każdy użytkownik, który przejdzie do późniejszych

etapów, musi być autoryzowany. Jeśli użytkownik o niskich uprawnieniach przejdzie bezpośrednio do późniejszego etapu, może mieć do niego dostęp bez żadnych ograniczeń.

Przykład 4: Rolowanie własnego ubezpieczenia

Autorzy napotkali tę lukę logiczną w aplikacji internetowej wdrożonej przez firmę świadczącą usługi finansowe.

Funkcjonalność

Aplikacja umożliwiała użytkownikom uzyskanie wyceny ubezpieczenia oraz, w razie potrzeby, wypełnienie i złożenie wniosku ubezpieczeniowego online. Proces został rozłożony na kilkanaście etapów:

* W pierwszym etapie wnioskodawca podał podstawowe informacje i określił preferowaną składkę miesięczną lub wartość ubezpieczenia, na jaką chciał. Aplikacja oferowała wycenę, obliczając wartość, której nie określił wnioskodawca.

* Na kilku etapach wnioskodawca podał różne inne dane osobowe, w tym stan zdrowia, zawód i rozrywki.

* Ostatecznie wniosek został przekazany do ubezpieczyciela pracującego dla firmy ubezpieczeniowej. Korzystając z tej samej aplikacji internetowej, subemitent przejrzał szczegóły i zdecydował, czy zaakceptować wniosek w obecnej postaci, czy zmodyfikować początkową wycenę, aby odzwierciedlić dodatkowe ryzyko.

Na każdym z opisanych etapów aplikacja wykorzystywała wspólny komponent do przetwarzania każdego parametru przesłanych do niej danych użytkownika. Komponent ten przetwarzał wszystkie dane w każdym żądaniu POST na pary nazwa/wartość i aktualizował informacje o stanie z każdym odebranym elementem danych.

Założenie

Komponent, który przetwarzał dane dostarczone przez użytkownika, zakładał, że każde żądanie będzie zawierało tylko te parametry, których zażądano od użytkownika w odpowiednim formularzu HTML. Deweloperzy nie zastanawiali się, co by się stało, gdyby użytkownik podał parametry, o których podanie nie został poproszony.

Atak

Oczywiście założenie było błędne, ponieważ użytkownicy mogli przy każdym żądaniu podawać dowolne nazwy i wartości parametrów. W rezultacie podstawowa funkcjonalność aplikacji została uszkodzona na różne sposoby:

* Osoba atakująca może wykorzystać udostępniony składnik, aby ominąć sprawdzanie poprawności danych wejściowych po stronie serwera. Na każdym etapie procesu wyceny aplikacja przeprowadzała ścisłą walidację danych oczekiwanych na tym etapie i odrzucała te, które tej walidacji nie przeszły. Jednak udostępniony komponent aktualizował stan aplikacji o każdy parametr podany przez użytkownika. W związku z tym, jeśli osoba atakująca przekaże dane poza kolejnością, podając parę nazwa/wartość, której aplikacja oczekiwała na wcześniejszym etapie, dane te zostaną zaakceptowane i przetworzone bez przeprowadzania walidacji. Tak się złożyło, że ta możliwość utorowała drogę do ataku typu cross-site scripting, wymierzonego w subemitenta, który umożliwił złośliwemu użytkownikowi dostęp do danych osobowych innych wnioskodawców (patrz rozdział 12).

* Osoba atakująca może wykupić ubezpieczenie po dowolnej cenie. Na pierwszym etapie procesu wyceny wnioskodawca określił albo preferowaną składkę miesięczną, albo wartość, którą chce ubezpieczyć, a aplikacja odpowiednio przeliczyła drugą pozycję. Jeśli jednak użytkownik podał nowe wartości dla jednego lub obu tych elementów na późniejszym etapie, stan aplikacji został zaktualizowany o te wartości. Przesyłając te parametry poza kolejnością, osoba atakująca może uzyskać wycenę ubezpieczenia o dowolnej wartości i dowolnej miesięcznej składce.

* Nie było kontroli dostępu do parametrów, które dany typ użytkownika mógłby podać. Kiedy subemitent przeglądał wypełniony wniosek, aktualizował różne elementy danych, w tym decyzję o przyjęciu. Dane te zostały przetworzone przez udostępniony komponent w taki sam sposób, jak dane dostarczone przez zwykłego użytkownika. Jeśli atakujący znał lub odgadywał nazwy parametrów używane podczas sprawdzania wniosku przez ubezpieczyciela, mógł po prostu je przesłać, akceptując w ten sposób własny wniosek bez faktycznego gwarantowania.

KROKI HACKOWANIA

Luki w tej aplikacji były fundamentalne dla jej bezpieczeństwa, ale żadna z nich nie zostałaby zidentyfikowana przez atakującego, który po prostu przechwyciłby żądania przeglądarki i zmodyfikował przesyłane wartości parametrów.

1. Zawsze, gdy aplikacja realizuje kluczowe działanie na wielu etapach, należy wziąć parametry, które są przesyłane na jednym etapie procesu i spróbować przesłać je do innego etapu. Jeśli odpowiednie elementy danych są aktualizowane w stanie aplikacji, należy zbadać konsekwencje takiego zachowania, aby określić, czy można je wykorzystać do przeprowadzenia złośliwych działań, jak w poprzednich trzech przykładach.

2. Jeżeli aplikacja ma zaimplementowaną funkcjonalność, dzięki której różne kategorie użytkowników mogą aktualizować lub wykonywać inne działania na wspólnym zbiorze danych, należy przejść przez proces z wykorzystaniem każdego typu użytkownika i obserwować podane parametry. Tam, gdzie różni użytkownicy zwykle przesyłają różne parametry, weź każdy parametr przesłany przez jednego użytkownika i spróbuj przesłać go jako inny użytkownik. Jeśli parametr zostanie zaakceptowany i przetworzony jako ten użytkownik, zbadaj implikacje tego zachowania zgodnie z wcześniejszym opisem.

Przykład 5: Włamanie do banku

Autorzy napotkali tę lukę logiczną w aplikacji internetowej wdrożonej przez dużą firmę świadczącą usługi finansowe.

Funkcjonalność

Aplikacja umożliwiła rejestrację dotychczasowym klientom, którzy nie korzystali jeszcze z aplikacji online. Nowi użytkownicy musieli podać podstawowe dane osobowe, aby zapewnić pewien stopień pewności co do swojej tożsamości. Informacje te obejmowały imię i nazwisko, adres i datę urodzenia, ale nie zawierały żadnych tajemnic, takich jak istniejące hasło lub kod PIN. Po prawidłowym wprowadzeniu tych informacji aplikacja przekazywała żądanie rejestracji do systemów zaplecza w celu przetworzenia. Pakiet informacyjny został wysłany na zarejestrowany adres domowy użytkownika. Pakiet ten zawierał instrukcję aktywacji jej dostępu online poprzez telefon do call center firmy, a także jednorazowe hasło do użycia przy pierwszym logowaniu do aplikacji.

Założenie

Twórcy aplikacji uważali, że mechanizm ten zapewnia solidną ochronę przed nieautoryzowanym dostępem do aplikacji. Mechanizm zaimplementował trzy poziomy ochrony:

* Z góry wymagana była niewielka ilość danych osobowych, aby powstrzymać złośliwego atakującego lub złośliwego użytkownika przed próbą zainicjowania procesu rejestracji w imieniu innych użytkowników.

* Proces polegał na przesłaniu tajnego klucza poza pasmem na zarejestrowany adres domowy klienta. Osoba atakująca musiałaby mieć dostęp do osobistej poczty ofiary.

* Klient musiał zadzwonić do call center i uwierzytelnić się tam w zwykły sposób, na podstawie danych osobowych i wybranych cyfr z PIN-u.

Ten projekt był rzeczywiście solidny. Błąd logiczny tkwił w uzupełnieniu mechanizmu. Twórcy wdrażający mechanizm rejestracji potrzebowali sposobu na przechowywanie danych osobowych podanych przez użytkownika i skorelowanie ich z unikalną tożsamością klienta w bazie danych firmy. Chcąc ponownie wykorzystać istniejący kod, natknęli się na następującą klasę, która wydawała się służyć ich celom:

```
class CCustomer

{

String firstName;

String lastName;

CDoB dob;

CAddress homeAddress;

long custNumber;

...
```

Po przechwyceniu informacji o użytkowniku obiekt ten został utworzony, wypełniony dostarczonymi informacjami i zapisany w sesji użytkownika. Następnie aplikacja weryfikowała dane użytkownika i, jeśli były prawidłowe, pobierała unikalny numer klienta, który był używany we wszystkich systemach firmy. Numer ten został dodany do obiektu wraz z kilkoma innymi przydatnymi informacjami o użytkowniku. Obiekt był następnie przekazywany do odpowiedniego systemu zaplecza w celu przetworzenia żądania rejestracji. Twórcy założyli, że użycie tego komponentu kodu jest nieszkodliwe i nie spowoduje problemów z bezpieczeństwem. Jednak założenie było błędne, co miało poważne konsekwencje.

Atak

Ten sam składnik kodu, który został włączony do funkcji rejestracji, był również używany w innym miejscu aplikacji, w tym w ramach podstawowej funkcjonalności. Dało to uwierzytelnionym użytkownikom dostęp do szczegółów konta, wyciągów, transferów środków i innych informacji. Gdy zarejestrowany użytkownik pomyślnie uwierzytelnił się w aplikacji, ten sam obiekt został utworzony i zapisany w jego sesji w celu przechowywania kluczowych informacji o jego tożsamości. Większość funkcjonalności w aplikacji odwoływała się do informacji zawartych w tym obiekcie w celu wykonywania swoich działań. Na przykład dane konta prezentowane użytkownikowi na jego stronie głównej zostały wygenerowane na podstawie unikalnego numeru klienta zawartego w tym obiekcie. Sposób, w jaki komponent kodu był już wykorzystany w aplikacji, oznaczał, że założenie programistów

było błędne, a sposób, w jaki go ponownie wykorzystali, faktycznie otworzył znaczącą lukę w zabezpieczeniach. Chociaż luka była poważna, jej wykrycie i wykorzystanie było w rzeczywistości stosunkowo subtelne. Dostęp do głównych funkcji aplikacji był chroniony przez kontrolę dostępu na kilku poziomach, a użytkownik musiał mieć w pełni uwierzytelnioną sesję, aby przejść przez te kontrole. Aby wykorzystać lukę logiczną, osoba atakująca musiała wykonać następujące kroki:

* Zaloguj się do aplikacji przy użyciu własnych ważnych poświadczeń konta.

* Korzystając z wynikowej uwierzytelnionej sesji, uzyskaj dostęp do funkcji rejestracji i prześlij dane osobowe innego klienta. Spowodowało to, że aplikacja nadpisała oryginalny obiekt CCustomer w sesji atakującego nowym obiektem odnoszącym się do docelowego klienta.

* Wróć do głównej funkcjonalności aplikacji i uzyskaj dostęp do konta innego klienta.

Luka tego rodzaju nie jest łatwa do wykrycia podczas badania aplikacji z perspektywy czarnej skrzynki. Jednak jest to również trudne do zidentyfikowania podczas przeglądania lub pisania rzeczywistego kodu źródłowego. Bez jasnego zrozumienia aplikacji jako całości i tego, jak różne komponenty są używane w różnych obszarach, błędne założenie przyjęte przez programistów może nie być oczywiste. Oczywiście przejrzyście skomentowany kod źródłowy i dokumentacja projektowa zmniejszyłyby prawdopodobieństwo wprowadzenia lub pozostania wykrycia takiej wady.

KROKI HACKOWANIA

1. W złożonej aplikacji obejmującej poziomą lub pionową segregację uprawnień spróbuj zlokalizować wszelkie przypadki, w których indywidualny użytkownik może zgromadzić w ramach swojej sesji stan, który w jakiś sposób odnosi się do jego tożsamości.

2. Spróbuj przejść przez jeden obszar funkcjonalności, a następnie przełącz się do niezwiązanego obszaru, aby określić, czy jakiekolwiek zgromadzone informacje o stanie mają wpływ na zachowanie aplikacji.

Przykład 6: Pokonanie limitu biznesowego

Autorzy napotkali tę lukę logiczną w internetowej aplikacji do planowania zasobów przedsiębiorstwa używanej w firmie produkcyjnej.

Funkcjonalność

Personel finansowy mógłby wykonywać przelewy środków między różnymi rachunkami bankowymi należącymi do firmy oraz jej kluczowych klientów i dostawców. W ramach zabezpieczenia przed oszustwami aplikacja uniemożliwiła większości użytkowników przetwarzanie przelewów o wartości większej niż 10 000 USD. Każdy większy transfer wymagał zatwierdzenia przez kierownika wyższego szczebla.

Założenie

Kod odpowiedzialny za wdrożenie tej kontroli w aplikacji był prosty:

```
bool CAuthCheck::RequiresApproval(int amount)

{

if (amount <= m_apprThreshold)

return false;
```

else return true;

}

Twórcy założyli, że ta przejrzysta kontrola jest kuloodporna. Żadna transakcja opiewająca na kwotę większą niż skonfigurowany próg nie może nigdy ujść wymogowi wtórnego zatwierdzenia.

Atak

Założenie twórców było błędne, ponieważ przeoczyli możliwość, że użytkownik spróbuje przetworzyć przelew na kwotę ujemną. Każda liczba ujemna wyeliminowałaby test zatwierdzający, ponieważ jest mniejsza niż próg. Jednak moduł bankowy aplikacji akceptował przelewy ujemne i po prostu przetwarzał je jako przelewy dodatnie w przeciwnym kierunku. Dlatego każdy użytkownik, który chciał przelać 20 000 USD z konta A na konto B, mógł po prostu zainicjować przelew –20 000 USD z konta B na konto A, co miało ten sam skutek i nie wymagało zatwierdzenia. Zabezpieczenia przed oszustwami wbudowane w aplikację można łatwo ominąć!

UWAGA: Wiele rodzajów aplikacji internetowych stosuje ograniczenia liczbowe w swojej logice biznesowej:

* Aplikacja do sprzedaży detalicznej może uniemożliwić użytkownikowi zamówienie większej liczby sztuk niż liczba jednostek dostępnych w magazynie.

* Aplikacja bankowa może uniemożliwić użytkownikowi dokonywanie płatności rachunków, które przekraczają jego bieżące saldo na koncie.

* Aplikacja ubezpieczeniowa może dostosować swoje oferty w oparciu o progi wiekowe.

Znalezienie sposobu na pokonanie takich ograniczeń często nie oznacza naruszenia bezpieczeństwa samej aplikacji. Może to jednak mieć poważne konsekwencje biznesowe i stanowić naruszenie kontroli, których egzekwowania właściciel wymaga od aplikacji. Najbardziej oczywiste luki tego rodzaju są często wykrywane podczas testów akceptacji użytkownika, które zwykle mają miejsce przed uruchomieniem aplikacji. Jednak mogą pozostać bardziej subtelne przejawy problemu, szczególnie w przypadku manipulowania ukrytymi parametrami.

KROKI HACKOWANIA

Pierwszym krokiem w próbie przekroczenia limitu biznesowego jest zrozumienie, jakie znaki są akceptowane w ramach odpowiednich danych wejściowych, które kontrolujesz.

1. Spróbuj wprowadzić wartości ujemne i sprawdź, czy aplikacja je akceptuje i przetwarza w oczekiwany sposób.

2. Może być konieczne wykonanie kilku kroków w celu zaprojektowania zmiany stanu aplikacji, którą można wykorzystać do użytecznego celu. Na przykład może być potrzebnych kilka przelewów między kontami, dopóki nie zostanie naliczone odpowiednie saldo, które można faktycznie pobrać.

Przykład 7: Oszukiwanie przy hurtowych rabatach

Autorzy napotkali tę lukę logiczną w detalicznej aplikacji dostawcy oprogramowania.

Funkcjonalność

Aplikacja umożliwiała użytkownikom zamawianie produktów oprogramowania i kwalifikowanie się do rabatów hurtowych w przypadku zakupu odpowiedniego pakietu elementów. Na przykład

użytkownicy, którzy zakupili rozwiązanie antywirusowe, zaporę osobistą i oprogramowanie antyspamowe mieli prawo do 25% rabatu od poszczególnych cen

Założenie

Gdy użytkownik dodał element oprogramowania do swojego koszyka, aplikacja stosowała różne reguły w celu ustalenia, czy wybrany przez niego pakiet zakupów uprawnia go do zniżki. Jeśli tak, ceny odpowiednich artykułów w koszyku zostały dostosowane zgodnie z rabatem. Twórcy założyli, że użytkownik kupi wybrany pakiet i tym samym będzie uprawniony do zniżki.

Atak

Założenie twórców jest dość ewidentnie błędne, ponieważ ignoruje fakt, że użytkownicy mogą usuwać pozycje ze swoich koszyków po ich dodaniu. Sprytny użytkownik mógłby dodać do swojego koszyka duże ilości każdego pojedynczego produktu oferowanego przez sprzedawcę, aby uzyskać maksymalne możliwe rabaty hurtowe. Po zastosowaniu rabatów na produkty w jego koszyku mógł usunąć niepotrzebne produkty i nadal otrzymywać rabaty na pozostałe produkty.

KROKI HACKOWANIA

1. W każdej sytuacji, w której ceny lub inne wrażliwe wartości są korygowane na podstawie kryteriów określonych przez dane lub działania kontrolowane przez użytkownika, najpierw należy zrozumieć algorytmy używane przez aplikację oraz punkt w jej logice, w którym dokonywane są korekty. Określ, czy te korekty są dokonywane jednorazowo, czy są korygowane w odpowiedzi na dalsze działania użytkownika.

2. Myśl z wyobraźnią. Spróbuj znaleźć sposób na manipulację zachowaniem aplikacji, aby wprowadzić ją w stan, w którym zastosowane przez nią korekty nie odpowiadają pierwotnym kryteriom zamierzonym przez jej projektantów. W najbardziej oczywistym przypadku, jak właśnie opisano, może to po prostu polegać na usunięciu pozycji z koszyka po zastosowaniu rabatu.

Przykład 8: Ucieczka przed ucieczką

Autorzy napotkali tę lukę logiczną w różnych aplikacjach internetowych, w tym w interfejsie administrowania siecią używanym przez produkt do wykrywania włamań do sieci.

Funkcjonalność

Projektanci aplikacji postanowili zaimplementować pewną funkcjonalność polegającą na przekazywaniu kontrolowanych przez użytkownika danych wejściowych jako argumentu do polecenia systemu operacyjnego. Twórcy aplikacji zrozumieli nieodłączne ryzyko związane z tego rodzaju operacjami (patrz rozdział 9) i postanowili bronić się przed tymi zagrożeniami, oczyszczając wszelkie potencjalnie złośliwe znaki w danych wejściowych użytkownika. Wszelkie wystąpienia następujących elementów zostaną zmienione za pomocą znaku ukośnika odwrotnego:

; | & < > ‘ spacja i znak nowej linii

Ucieczka danych w ten sposób powoduje, że interpreter poleceń powłoki traktuje odpowiednie znaki jako część argumentu przekazywanego do wywołanego polecenia, a nie jako metaznaki powłoki. Takich metaznaków można użyć do wstrzyknięcia dodatkowych poleceń lub argumentów, przekierowania danych wyjściowych i tak dalej.

Założenie

Deweloperzy byli pewni, że opracowali solidną obronę przed atakami polegającymi na wstrzykiwaniu poleceń. Przeprowadzili burzę mózgów na temat każdej możliwej postaci, która mogłaby pomóc atakującemu i upewnili się, że wszyscy zostali odpowiednio uciekli, a tym samym bezpieczni.

Atak

Twórcy zapomnieli uciec od samej postaci ucieczki. Znak odwrotnego ukośnika zwykle nie jest bezpośrednio użyteczny dla atakującego, gdy wykorzystując prosty błąd wstrzykiwania poleceń. Dlatego twórcy nie zidentyfikowali go jako potencjalnie złośliwego. Jednakże, nie mogąc z niego uciec, zapewnili atakującemu środki do pokonania ich mechanizmu odkażającego. Załóżmy, że osoba atakująca dostarcza następujące dane wejściowe do podatnej na ataki funkcji:

foo\;ls

Aplikacja stosuje odpowiednią ucieczkę, jak opisano wcześniej, więc wejście atakującego ma postać:

foo\\;ls

Kiedy te dane są przekazywane jako argument do polecenia systemu operacyjnego, interpreter powłoki traktuje pierwszy ukośnik odwrotny jako znak ucieczki. Dlatego traktuje drugi ukośnik odwrotny jako dosłowny ukośnik odwrotny — nie jako znak zmiany znaczenia, ale jako część samego argumentu. Następnie napotyka średnik, który najwyraźniej nie jest znakiem ucieczki. Traktuje to jako separator poleceń i dlatego wykonuje wstrzyknięte polecenie dostarczone przez atakującego.

KROKI HACKOWANIA

Za każdym razem, gdy sprawdzasz aplikację pod kątem wstrzykiwania poleceń i innych błędów, po próbie wstawienia odpowiednich metaznaków do danych, które kontrolujesz, zawsze spróbuj umieścić ukośnik odwrotny bezpośrednio przed każdym takim znakiem, aby sprawdzić opisaną wadę logiczną.

UWAGA: Tę samą wadę można znaleźć w niektórych zabezpieczeniach przed atakami typu cross-site scripting . Gdy dane wejściowe dostarczone przez użytkownika są kopiowane bezpośrednio do wartości zmiennej łańcuchowej w fragmencie kodu JavaScript, ta wartość jest ujęta w cudzysłowy. Aby bronić się przed skryptami między witrynami, wiele aplikacji używa odwrotnych ukośników, aby uniknąć cudzysłowów pojawiających się w danych wejściowych użytkownika. Jeśli jednak sam znak ukośnika odwrotnego nie jest znakiem ucieczki, osoba atakująca może przesłać \', aby wyrwać się z łańcucha iw ten sposób przejąć kontrolę nad skryptem. Dokładnie ten błąd został znaleziony we wczesnych wersjach frameworka Ruby On Rails w funkcji escape_javascript.

Przykład 9: Unieważnienie walidacji danych wejściowych

Autorzy napotkali tę lukę logiczną w aplikacji internetowej używanej w witrynie e-commerce. Warianty można znaleźć w wielu innych zastosowaniach.

Funkcjonalność

Aplikacja zawierała zestaw procedur sprawdzania poprawności danych wejściowych w celu ochrony przed różnego rodzaju atakami. Dwa z tych mechanizmów obronnych to filtr iniekcji SQL i ogranicznik długości. Często zdarza się, że aplikacje próbują bronić się przed wstrzyknięciem kodu SQL, unikając pojedynczych cudzysłowów, które pojawiają się w danych wejściowych użytkownika opartych na łańcuchach (i odrzucając wszystkie, które pojawiają się w danych liczbowych). Jak opisano w rozdziale 9, dwa pojedyncze cudzysłowy razem tworzą sekwencję ucieczki, która reprezentuje jeden dosłowny pojedynczy cudzysłów, który baza danych interpretuje jako dane w ciągu ujętym w cudzysłowy, a nie kończący ciąg znaków. Dlatego wielu programistów uważa, że podwojenie pojedynczych cudzysłowów

w danych wejściowych dostarczonych przez użytkownika zapobiegnie wszelkim atakom polegającym na iniekcji SQL. Do wszystkich danych wejściowych zastosowano ogranicznik długości, dzięki czemu żadna zmienna podana przez użytkownika nie była dłuższa niż 128 znaków. Osiągnięto to poprzez obcięcie dowolnych zmiennych do 128 znaków.

Założenie

Założono, że zarówno filtr wstrzykiwania SQL, jak i obcinanie długości są pożądanymi zabezpieczeniami z punktu widzenia bezpieczeństwa, dlatego należy zastosować oba.

Atak

Ochrona przed iniekcją SQL polega na podwojeniu wszelkich cudzysłowów pojawiających się w danych wprowadzonych przez użytkownika, dzięki czemu w każdej parze cudzysłowów pierwszy cudzysłów działa jako znak zmiany znaczenia dla drugiego. Jednak programiści nie zastanawiali się, co stałoby się z oczyszczonymi danymi wejściowymi, gdyby zostały następnie przekazane funkcji obcinania. Przypomnij sobie przykład wstrzykiwania kodu SQL w funkcji logowania z rozdziału 9. Załóżmy, że aplikacja podwaja pojedyncze cudzysłowy zawarte w danych wprowadzanych przez użytkownika, a następnie narzuca ograniczenie długości danych, obcinając je do 128 znaków. Podanie tej nazwy użytkownika:

admin'--

teraz skutkuje następującym zapytaniem, które nie omija logowania:

SELECT * FROM users WHERE username = ‘admin’’--’ and password = ‘’

Jeśli jednak prześlesz następującą nazwę użytkownika (zawierającą 127 a, po których następuje pojedynczy cudzysłów):

aaaaaaaa[...]aaaaaaaaaaaa

aplikacja najpierw podwaja pojedynczy cudzysłów, a następnie skraca łańcuch do 128 znaków, przywracając wartość wprowadzoną przez użytkownika. Powoduje to błąd bazy danych, ponieważ do zapytania wprowadzono dodatkowy pojedynczy cudzysłów bez poprawiania otaczającej składni. Jeśli teraz podasz również hasło:

or 1=1--

aplikacja wykonuje następujące zapytanie, które udaje się ominąć logowanie:

SELECT * FROM users WHERE username = ‘aaaaaaaa[...]aaaaaaaaaaa’’ and

password = ‘or 1=1--’

Podwójny cudzysłów na końcu ciągu znaków a jest interpretowany jako cudzysłów ze zmianą znaczenia, a zatem jako część danych zapytania. Ten ciąg skutecznie kontynuuje się aż do następnego pojedynczego cudzysłowu, który w pierwotnym zapytaniu oznaczał początek wartości hasła podanej przez użytkownika. Tak więc rzeczywista nazwa użytkownika, którą rozumie baza danych, to dosłowne dane łańcuchowe pokazane tutaj:

aaaaaaaa[...]aaaaaaaaa i hasło =

W związku z tym wszystko, co nastąpi później, jest interpretowane jako część samego zapytania i może zostać spreparowane w celu ingerencji w logikę zapytania.

WSKAZÓWKA: Możesz przetestować ten typ luki, nie wiedząc dokładnie, jaki limit długości jest nakładany, przesyłając po kolei dwa długie ciągi o następującej postaci:

'''''''''''''''''''''''''''''''''''' i tak dalej

a’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’ i tak dalej

i określenie, czy wystąpił błąd. Każde obcięcie wejścia ze znakiem ucieczki nastąpi po parzystej lub nieparzystej liczbie znaków. Niezależnie od tego, jaka jest możliwość, jeden z poprzedzających ciągów spowoduje wstawienie do zapytania nieparzystej liczby pojedynczych cudzysłowów, co spowoduje nieprawidłową składnię.

KROKI HACKOWANIA

Zanotuj wszystkie przypadki, w których aplikacja modyfikuje dane wprowadzane przez użytkownika, w szczególności je obcinając, usuwając dane, kodując lub dekodując. W przypadku zaobserwowanych przypadków określ, czy można wymyślić złośliwy ciąg znaków:

1. Jeśli dane są usuwane raz (nierekurencyjnie), określ, czy możesz przesłać ciąg, który to kompensuje. Na przykład, jeśli aplikacja filtruje słowa kluczowe SQL, takie jak SELECT, prześlij SELSELECTECT i zobacz, czy wynikowe filtrowanie usunie wewnętrzny podłańcuch SELECT, pozostawiając słowo SELECT.

2. Jeśli walidacja danych odbywa się w ustalonej kolejności i jeden lub więcej procesów walidacji modyfikuje dane, określ, czy można to wykorzystać do pokonania jednego z wcześniejszych etapów walidacji. Na przykład, jeśli aplikacja przeprowadza dekodowanie adresów URL, a następnie usuwa złośliwe dane, takie jak tag <script>, można temu zaradzić za pomocą ciągów znaków, takich jak:

%<skrypt>3cscript%<skrypt>3ealert(1)%<skrypt>3c/

skrypt%<skrypt>3e

UWAGA: Filtry cross-site scripting często w sposób niezamierzony usuwają wszystkie dane występujące między parami znaczników HTML, takimi jak <tag1>aaaaa</tag1>. Są one często podatne na tego typu ataki.

Przykład 10: Nadużywanie funkcji wyszukiwania

Autorzy napotkali tę lukę logiczną w aplikacji zapewniającej subskrypcyjny dostęp do wiadomości i informacji finansowych. Ta sama luka została później odkryta w dwóch zupełnie niezwiązanych ze sobą aplikacjach, co ilustruje subtelną i wszechobecną naturę wielu błędów logicznych.

Funkcjonalność

Aplikacja zapewniała dostęp do ogromnego archiwum informacji historycznych i bieżących, w tym raportów i rozliczeń spółek, komunikatów prasowych, analiz rynkowych i tym podobnych. Większość tych informacji była dostępna tylko dla płacących abonentów. Aplikacja zapewniała wydajną i precyzyjną funkcję wyszukiwania, do której dostęp mieli wszyscy użytkownicy. Gdy anonimowy użytkownik wykonał zapytanie, funkcja wyszukiwania zwróciła łącza do wszystkich dokumentów pasujących do zapytania. Jednak użytkownik musiał wykupić subskrypcję, aby pobrać dowolny z faktycznie chronionych dokumentów zwróconych przez jego zapytanie. Właściciele aplikacji uznali to zachowanie za użyteczną taktykę marketingową.

Założenie

Projektant aplikacji założył, że użytkownicy nie mogą korzystać z funkcji wyszukiwania w celu wydobycia przydatnych informacji bez płacenia za nie. Tytuły dokumentów wymienione w wynikach wyszukiwania były zazwyczaj tajemnicze, na przykład „Wyniki roczne 2010", „Komunikat prasowy z dnia 08-03-2011" i tak dalej.

Atak

Ponieważ funkcja wyszukiwania wskazywała, ile dokumentów pasuje do danego zapytania, przebiegły użytkownik mógł zadać dużą liczbę zapytań i użyć wnioskowania, aby wyodrębnić z funkcji wyszukiwania informacje, za które normalnie trzeba by zapłacić. Na przykład następujące zapytania mogą służyć do zerowania zawartości pojedynczego chronionego dokumentu:

wahh consulting

>> 276 matches

wahh consulting "Press Release 08-03-2011" merger

>> 0 matches

wahh consulting "Press Release 08-03-2011" share issue

>> 0 matches

wahh consulting "Press Release 08-03-2011" dividend

>> 0 matches

wahh consulting "Press Release 08-03-2011" takeover

>> 1 match

wahh consulting "Press Release 08-03-2011" takeover haxors inc

>> 0 matches

wahh consulting "Press Release 08-03-2011" takeover uberleet ltd

>> 0 matches

wahh consulting "Press Release 08-03-2011" takeover script kiddy corp

>> 0 matches

wahh consulting "Press Release 08-03-2011" takeover ngs

>> 1 match

wahh consulting "Press Release 08-03-2011" takeover ngs announced

>> 0 matches

wahh consulting "Press Release 08-03-2011" takeover ngs cancelled

>> 0 matches

wahh consulting "Press Release 08-03-2011" takeover ngs completed

>> 1 match

Chociaż użytkownik nie może przeglądać samego dokumentu, przy wystarczającej wyobraźni i użyciu skryptowych żądań, może być w stanie zbudować dość dokładne zrozumienie jego zawartości.

WSKAZÓWKA: W niektórych sytuacjach możliwość wyłudzenia informacji za pomocą funkcji wyszukiwania w ten sposób może mieć kluczowe znaczenie dla bezpieczeństwa samej aplikacji, skutecznie ujawniając szczegóły funkcji administracyjnych, haseł i używanych technologii.

WSKAZÓWKA: Ta technika okazała się skutecznym atakiem na wewnętrzne oprogramowanie do zarządzania dokumentami. Autorzy wykorzystali tę technikę do brutalnego wymuszenia hasła klucza z pliku konfiguracyjnego, który był przechowywany na wiki. Ponieważ wiki zwróciło trafienie, jeśli wyszukiwany ciąg pojawił się w dowolnym miejscu na stronie (zamiast dopasowywania całych słów), możliwe było brutalne wymuszenie hasła litera po literze, wyszukując:

Hasło=A

Hasło=B

Hasło=BA

...

Przykład 11: Snarfingowanie komunikatów debugowania

Autorzy napotkali tę lukę logiczną w aplikacji internetowej używanej przez firmę świadczącą usługi finansowe.

Funkcjonalność

Aplikacja została niedawno wdrożona. Podobnie jak wiele nowych programów, nadal zawierało szereg błędów związanych z funkcjonalnością. Sporadycznie różne operacje kończyły się niepowodzeniem w nieprzewidywalny sposób, a użytkownicy otrzymywali komunikat o błędzie. Aby ułatwić badanie błędów, programiści postanowili zawrzeć w tych wiadomościach szczegółowe, obszerne informacje, w tym następujące szczegóły:

* Tożsamość użytkownika

* Token dla bieżącej sesji

* Dostęp do adresu URL

* Wszystkie parametry dostarczone z żądaniem, które wygenerowało błąd

Generowanie tych komunikatów okazało się przydatne, gdy personel pomocy technicznej próbował zbadać i naprawić awarie systemu. Pomagali także usunąć pozostałe błędy funkcjonalne.

Założenie

Pomimo zwykłych ostrzeżeń ze strony doradców ds. bezpieczeństwa, że szczegółowe komunikaty debugowania tego rodzaju mogą potencjalnie zostać wykorzystane przez atakującego, programiści uznali, że nie otwierają żadnej luki w zabezpieczeniach. Użytkownik mógł łatwo uzyskać wszystkie informacje zawarte w komunikacie debugowania, sprawdzając żądania i odpowiedzi przetwarzane przez jej przeglądarkę. Komunikaty nie zawierały żadnych szczegółów dotyczących rzeczywistej awarii, takich jak ślady stosu, więc prawdopodobnie nie były pomocne w formułowaniu ataku na aplikację.

Atak

Pomimo ich rozumowania na temat treści komunikatów debugowania, założenie programistów było błędne z powodu błędów, które popełnili podczas implementacji tworzenia komunikatów debugowania. Gdy wystąpił błąd, składnik aplikacji zbierał wszystkie pliki wymaganych informacji i zapisał je. Użytkownik otrzymał przekierowanie HTTP do adresu URL, który wyświetlał te zapisane informacje. Problem polegał na tym, że przechowywanie informacji debugowania przez aplikację i dostęp użytkownika do komunikatu o błędzie nie były oparte na sesjach. Zamiast tego informacje debugowania były przechowywane w kontenerze statycznym, a adres URL komunikatu o błędzie zawsze wyświetlał informacje, które były ostatnio umieszczane w tym kontenerze. Deweloperzy założyli, że użytkownicy podążający za przekierowaniem zobaczą tylko informacje debugowania dotyczące ich błędu. W rzeczywistości w tej sytuacji zwykli użytkownicy czasami otrzymywali informacje debugowania dotyczące błędu innego użytkownika, ponieważ te dwa błędy wystąpiły prawie jednocześnie. Ale oprócz pytań o bezpieczeństwo nici (patrz następny przykład), nie był to zwykły wyścig. Osoba atakująca, która odkryła, jak działa mechanizm błędu, mogłaby po prostu wielokrotnie sondować adres URL wiadomości i rejestrować wyniki za każdym razem, gdy uległy one zmianie. W ciągu kilku godzin ten dziennik zawierałby poufne dane dotyczące wielu użytkowników aplikacji:

* Zestaw nazw użytkowników, których można użyć w ataku polegającym na odgadywaniu hasła

* Zestaw tokenów sesji, których można użyć do przejęcia sesji

* Zestaw danych wprowadzonych przez użytkownika, który może zawierać hasła i inne poufne elementy

Mechanizm błędu stanowił zatem krytyczne zagrożenie bezpieczeństwa. Ponieważ użytkownicy administracyjni czasami otrzymywali te szczegółowe komunikaty o błędach, osoba atakująca monitorująca komunikaty o błędach szybko uzyskałaby informacje wystarczające do skompromitowania całej aplikacji.

KROKI HACKOWANIA

1. Aby wykryć tego rodzaju lukę, najpierw skataloguj wszystkie nietypowe zdarzenia i warunki, które mogą zostać wygenerowane i które obejmują interesujące informacje specyficzne dla użytkownika zwracane do przeglądarki w nietypowy sposób, na przykład komunikat o błędzie debugowania.

2. Korzystając z aplikacji równolegle przez dwóch użytkowników, systematycznie konstruuj każdy warunek przy użyciu jednego lub obu użytkowników i określ, czy w każdym przypadku dotyczy to drugiego użytkownika.

Przykład 12: Wyścig z loginem

Ta usterka logiczna miała wpływ na kilka głównych aplikacji w niedawnej przeszłości.

Funkcjonalność

Aplikacja zaimplementowała solidny, wieloetapowy proces logowania, w którym użytkownicy musieli podać kilka różnych danych uwierzytelniających, aby uzyskać dostęp.

Założenie

Mechanizm uwierzytelniania był przedmiotem licznych przeglądów projektu i testów penetracyjnych. Właściciele byli przekonani, że nie istnieją żadne możliwe sposoby zaatakowania mechanizmu w celu uzyskania nieautoryzowanego dostępu.

Atak

W rzeczywistości mechanizm uwierzytelniania zawierał subtelną wadę. Czasami po zalogowaniu klient uzyskiwał dostęp do konta zupełnie innego użytkownika, co umożliwiało mu przeglądanie wszystkich danych finansowych tego użytkownika, a nawet dokonywanie płatności z konta innego użytkownika. Zachowanie aplikacji początkowo wydawało się przypadkowe: użytkownik nie wykonywał żadnych nietypowych działań w celu uzyskania nieautoryzowanego dostępu, a anomalia nie powtarzała się przy kolejnych logowaniach. Po pewnym dochodzeniu bank odkrył, że błąd występował, gdy dwóch różnych użytkowników logowało się do aplikacji dokładnie w tym samym momencie. Nie zdarzało się to przy każdej takiej okazji — tylko przy ich podzbiorze. Podstawową przyczyną było to, że aplikacja przez krótki czas przechowywała identyfikator klucza dotyczący każdego nowo uwierzytelnionego użytkownika w zmiennej statycznej (niesesyjnej). Po zapisaniu wartość tej zmiennej została odczytana chwilę później. Gdyby inny wątek (przetwarzający inny login) zapisał zmienną w tym momencie, wcześniejszy użytkownik wylądowałby w uwierzytelnionej sesji należącej do kolejnego użytkownika. Luka powstała w wyniku tego samego rodzaju błędu, co w opisanym wcześniej przykładzie z komunikatem o błędzie: aplikacja używała pamięci statycznej do przechowywania informacji, które powinny być przechowywane dla poszczególnych wątków lub sesji. Jednak obecny przykład jest znacznie bardziej subtelny do wykrycia i trudniejszy do wykorzystania, ponieważ nie można go wiarygodnie odtworzyć. Wady tego rodzaju są znane jako „warunki wyścigowe", ponieważ wiążą się z podatnością, która pojawia się na krótki okres czasu w pewnych określonych okolicznościach. Ponieważ luka istnieje tylko przez krótki czas, atakujący „ściga się", by ją wykorzystać, zanim aplikacja ponownie ją zamknie. W przypadkach, gdy osoba atakująca działa lokalnie w aplikacji, często możliwe jest zaprojektowanie dokładnych okoliczności, w których powstaje sytuacja wyścigu, i niezawodne wykorzystanie luki w dostępnym oknie. Jeśli atakujący znajduje się daleko od aplikacji, jest to zwykle znacznie trudniejsze do osiągnięcia. Zdalny atakujący, który zrozumiał naturę luki, mógł wymyślić atak, aby ją wykorzystać, używając skryptu do ciągłego logowania i sprawdzania szczegółów konta, do którego uzyskuje dostęp. Ale małe okienko czasu, w którym luka mogła zostać wykorzystana, oznaczał, że wymagana byłaby ogromna liczba żądań. Nic dziwnego, że warunki wyścigu nie zostały odkryte podczas normalnych testów penetracyjnych. Warunki, w jakich powstał, zaistniały dopiero wtedy, gdy aplikacja zyskała na tyle dużą bazę użytkowników, że zdarzały się przypadkowe anomalie zgłaszane przez klientów. Jednak dokładny przegląd kodu logiki uwierzytelniania i zarządzania sesją pozwoliłby zidentyfikować problem.

KROKI HACKOWANIA

Wykonywanie zdalnych testów czarnej skrzynki pod kątem subtelnych problemów związanych z bezpieczeństwem wątków tego rodzaju nie jest proste. Należy to traktować jako wyspecjalizowane przedsięwzięcie, prawdopodobnie niezbędne tylko w najbardziej krytycznych pod względem bezpieczeństwa aplikacjach.

1. Celuj w wybrane elementy kluczowych funkcjonalności, takie jak mechanizmy logowania, funkcje zmiany hasła i procesy transferu środków.

2. Dla każdej testowanej funkcji zidentyfikuj pojedyncze żądanie lub niewielką liczbę żądań, które dany użytkownik może wykorzystać do wykonania jednej akcji. Znajdź również najprostszy sposób potwierdzenia wyniku akcji, np. sprawdzenie, czy logowanie danego użytkownika spowodowało dostęp do informacji o koncie tej osoby.

3. Korzystając z kilku maszyn o wysokich parametrach, uzyskujących dostęp do aplikacji z różnych lokalizacji sieciowych, przygotuj skrypt ataku, aby wielokrotnie wykonywać tę samą akcję w imieniu kilku różnych użytkowników. Potwierdź, czy każde działanie ma oczekiwany rezultat.

4. Przygotuj się na dużą liczbę fałszywych trafień. W zależności od skali infrastruktury wspierającej aplikację czynność ta może równie dobrze równać się testowi obciążeniowemu instalacji. Anomalie mogą wystąpić z przyczyn, które nie mają nic wspólnego z bezpieczeństwem.

Unikanie błędów logicznych

Tak jak nie ma unikalnej sygnatury, za pomocą której można zidentyfikować błędy logiczne w aplikacjach internetowych, nie ma też złotego środka, który Cię ochroni. Na przykład nie ma odpowiednika prostej porady dotyczącej korzystania z bezpiecznej alternatywy dla niebezpiecznego interfejsu API. Niemniej jednak można zastosować szereg dobrych praktyk, które znacznie zmniejszą ryzyko wystąpienia błędów logicznych w Twoich aplikacjach:

* Upewnij się, że każdy aspekt projektu aplikacji jest wyraźnie udokumentowany i wystarczająco szczegółowy, aby osoba z zewnątrz zrozumiała każde założenie przyjęte przez projektanta. Wszystkie takie założenia powinny być - wyraźnie zapisane w dokumentacji projektowej.

* Nakaz, aby cały kod źródłowy był wyraźnie opatrzony komentarzem i zawierał następujące informacje:

* Cel i zamierzone zastosowania każdego składnika kodu.

* Założenia przyjęte przez każdy komponent na temat wszystkiego, co jest poza jego bezpośrednią kontrolą.

* Odwołania do całego kodu klienta korzystającego z komponentu. Jasna dokumentacja w tym zakresie mogłaby zapobiec błędowi logicznemu w funkcji rejestracji online. (Zauważ, że „klient” nie odnosi się tutaj do strony użytkownika relacji klient/serwer, ale do innego kodu, dla którego rozważany komponent jest bezpośrednią zależnością.)

* Podczas przeglądów projektu aplikacji pod kątem bezpieczeństwa zastanów się nad każdym założeniem przyjętym w projekcie i spróbuj wyobrazić sobie okoliczności, w których każde założenie może zostać naruszone. Skoncentruj się na wszelkich założonych warunkach, które mogą znajdować się pod kontrolą użytkowników aplikacji.

* Podczas przeglądów kodu skoncentrowanych na bezpieczeństwie pomyśl z boku o dwóch kluczowych obszarach: sposobach, w jakie aplikacja poradzi sobie z nieoczekiwanymi zachowaniami użytkowników oraz potencjalnymi skutkami ubocznymi wszelkich zależności i współdziałania między różnymi komponentami kodu i różnymi funkcjami aplikacji. W odniesieniu do konkretnych przykładów błędów logicznych, które opisaliśmy, można wyciągnąć kilka indywidualnych wniosków:

* Miej stale świadomość, że użytkownicy kontrolują każdy aspekt każdego żądania (patrz rozdział 1). Mogą uzyskiwać dostęp do funkcji wieloetapowych w dowolnej kolejności. Mogą podać parametry, o które aplikacja nie prosiła. Mogą pomijać niektóre parametry, a nie tylko ingerować w wartości parametrów.

* Kieruj wszystkimi decyzjami dotyczącymi tożsamości i statusu użytkownika na podstawie jego sesji (patrz rozdział 8). Nie zakładaj żadnych uprawnień użytkownika na podstawie jakiejkolwiek innej cechy żądania, w tym faktu, że w ogóle występuje.

* Wdrażając funkcje, które aktualizują dane sesji na podstawie danych wejściowych otrzymanych od użytkownika lub działań wykonanych przez użytkownika, należy uważnie rozważyć wpływ, jaki zaktualizowane dane mogą mieć na inne funkcje w aplikacji. Należy pamiętać, że nieoczekiwane efekty

uboczne mogą wystąpić w całkowicie niezwiązanych ze sobą funkcjach napisanych przez innego programistę lub nawet inny zespół programistów.

* Jeśli funkcja wyszukiwania może indeksować poufne dane, do których niektórzy użytkownicy nie mają uprawnień dostępu, upewnij się, że funkcja ta nie zapewnia tym użytkownikom żadnych możliwości wnioskowania o informacjach na podstawie wyników wyszukiwania. W razie potrzeby utrzymuj kilka indeksów wyszukiwania opartych na różnych poziomach uprawnień użytkownika lub przeprowadzaj dynamiczne przeszukiwanie repozytoriów informacji z uprawnieniami żądającego użytkownika.

* Zachowaj szczególną ostrożność przy wdrażaniu jakichkolwiek funkcji, które umożliwiają dowolnemu użytkownikowi usuwanie elementów ze ścieżki audytu. Należy również wziąć pod uwagę możliwy wpływ, jaki użytkownik o wysokich uprawnieniach może mieć na utworzenie innego użytkownika o tym samym poziomie uprawnień w aplikacjach poddanych intensywnemu audytowi i modelach podwójnej autoryzacji.

* Podczas przeprowadzania kontroli na podstawie liczbowych limitów i progów biznesowych należy przeprowadzić ścisłą kanonizację i weryfikację danych wszystkich danych wprowadzonych przez użytkownika przed ich przetworzeniem. Jeśli nie oczekuje się liczb ujemnych, wyraźnie odrzuć żądania, które je zawierają.

* Wdrażając rabaty oparte na wielkości zamówień, upewnij się, że zamówienia zostały sfinalizowane przed faktycznym zastosowaniem rabatu.

* Podczas ucieczki danych dostarczonych przez użytkownika przed przekazaniem ich do potencjalnie podatnego na ataki komponentu aplikacji, zawsze pamiętaj o zmianie samego znaku ucieczki, w przeciwnym razie cały mechanizm sprawdzania poprawności może zostać uszkodzony.

* Zawsze używaj odpowiedniego miejsca do przechowywania wszelkich danych odnoszących się do indywidualnego użytkownika — zarówno w sesji, jak iw profilu użytkownika.

Streszczenie

Atakowanie logiki aplikacji obejmuje połączenie systematycznego sondowania i myślenia bocznego. Opisaliśmy różne kluczowe kontrole, które zawsze należy przeprowadzać, aby przetestować zachowanie aplikacji w odpowiedzi na nieoczekiwane dane wejściowe. Obejmują one usuwanie parametrów z żądań, korzystanie z wymuszonego przeglądania w celu uzyskania dostępu do funkcji poza kolejnością oraz przesyłanie parametrów do różnych lokalizacji w aplikacji. Często sposób, w jaki aplikacja reaguje na te działania, wskazuje na wadliwe założenie, które można naruszyć, co może mieć szkodliwy wpływ. Oprócz tych podstawowych testów, najważniejszym wyzwaniem podczas szukania błędów logicznych jest próba dostania się do umysłów programistów. Musisz zrozumieć, co próbowali osiągnąć, jakie prawdopodobnie przyjęli założenia, jakie drogi na skróty prawdopodobnie wybrali i jakie błędy mogli popełnić. Wyobraź sobie, że pracujesz w napiętym terminie, martwiąc się przede wszystkim funkcjonalnością, a nie bezpieczeństwem, próbujesz dodać nową funkcję do istniejącej bazy kodu lub używasz słabo udokumentowanych interfejsów API napisanych przez kogoś innego. Co w takiej sytuacji zrobiłbyś źle i jak można to wykorzystać?

Pytania

1. Co to jest wymuszone przeglądanie i jakiego rodzaju luki w zabezpieczeniach można dzięki niemu zidentyfikować?

2. Aplikacja stosuje różne globalne filtry danych wprowadzanych przez użytkownika, zaprojektowane w celu zapobiegania różnym kategoriom ataków. Aby bronić się przed iniekcją SQL, podwaja wszystkie pojedyncze cudzysłowy, które pojawiają się w danych wejściowych użytkownika. Aby zapobiec atakom przepełnienia bufora na niektóre komponenty kodu natywnego, obcina zbyt długie elementy do rozsądnego limitu. Co może pójść nie tak z tymi filtrami?

3. Jakie kroki możesz podjąć, aby zbadać funkcję logowania pod kątem warunków otwarcia awaryjnego? (Opisz tyle różnych testów, ile możesz wymyślić).

4. Aplikacja bankowa implementuje wieloetapowy mechanizm logowania, który ma być wysoce niezawodny. W pierwszym etapie użytkownik podaje nazwę użytkownika i hasło. W drugim etapie użytkownik wprowadza zmienną alue na posiadanym tokenie fizycznym, a pierwotna nazwa użytkownika jest ponownie wprowadzana w ukrytym polu formularza.

Jaką lukę logiczną należy natychmiast sprawdzić?

5. Badasz aplikację pod kątem typowych kategorii luk w zabezpieczeniach, przesyłając spreparowane dane wejściowe. Często aplikacja zwraca obszerne komunikaty o błędach zawierające informacje dotyczące debugowania. Czasami komunikaty te dotyczą błędów generowanych przez innych użytkowników. Gdy tak się stanie, nie będziesz w stanie odtworzyć tego zachowania po raz drugi. Na jaki błąd logiczny może to wskazywać i jak należy postąpić?