MODUŁ 9:
Podstawy bezpieczeństwa aplikacji WWW

Cel modułu

Niniejszy moduł przedstawia istotne zagadnienia bezpieczeństwa aplikacji WWW i, zgodnie z naszym rozumieniem pojęcia takiej aplikacji, obejmuje problemy dotyczące części serwerowej, klienckiej, a także znajdującej się pomiędzy nimi infrastruktury komunikacyjnej. Specjaliści od bezpieczeństwa podkreślają, że powyższą listę należy rozszerzyć jeszcze o człowieka (jako użytkownika oraz programistę systemu), gdyż niezależnie od skuteczności rozwiązań technicznych, sumaryczne bezpieczeństwo jest silnie zależne od czynnika ludzkiego. Moduł rozpoczyna omówienie mechanizmów podtrzymywania stanu w aplikacjach WWW (element bliżej nieprezentowany wcześniej a istotny z punktu widzenia bezpieczeństwa). Następnie dokonamy przeglądu zagrożeń aplikacji WWW opierając się na rankingu OWASP[1].

Efekty kształcenia

Niniejsza lekcja służy osiągnięciu następujących efektów kształcenia.

Wiedza

Po ukończonym kursie student / studentka:

- Charakteryzuje mechanizmy podtrzymywania danych sesji użytkownika komunikującego się z witryną WWW,
- Przedstawia najważniejsze rodzaje zagrożeń dla aplikacji WWW oraz sposoby przeciwdziałania im.

Umiejętności

Po ukończonym kursie student / studentka:

- Dokonuje prostych sprawdzeń podatności aplikacji WWW na najważniejsze zagrożenia dla bezpieczeństwa i prywatności,
- Wykorzystuje funkcjonalność ramy aplikacji WWW wspierającą zapewnienie bezpieczeństwa aplikacji,
- Wykorzystuje w konstrukcji aplikacji WWW różnorodne mechanizmy podtrzymywania stanu interakcji klienta z aplikacją,
- Identyfikuje, poprzez analizę komunikacji klienta z serwerem, podtrzymywanie stanu interakcji klienta z aplikacją WWW.

Uzasadnienie

Zagadnienia bezpieczeństwa aplikacji WWW nie sposób przecenić. Aplikacja WWW stanowi bezprecedensowy rodzaj oprogramowania, który jest otwarty na przyjmowanie zleceń i parametrów z całego świata, a równocześnie dysponuje niekiedy newralgicznymi danymi i/lub połączeniami z szeregiem innych zasobów informatycznych danej organizacji. Sprawia to, że na programiście aplikacji WWW spoczywa bardzo wielka odpowiedzialność.

---

[1] Open Web Application Security Project (OWASP) https://www.owasp.org – organizacja non-profit popularyzująca wiedzę na temat doskonalenia bezpieczeństwa w oprogramowaniu.

Wprowadzenie

Na obecnym etapie realizacji kursu mamy już wgląd w kluczowe elementy aplikacji WWW, a mianowicie: część serwerową wraz ze środowiskiem interpretacji kodu mechanizmami ramy oraz bazą danych, infrastrukturę komunikacyjną związaną z protokołem HTTP, środowisko przeglądarki z jego udogodnieniami i technologiami programistycznymi warstwy klienta, a także – przetwarzane przez nie dokumenty znacznikowe oraz arkusze stylów. Ta wiedza będzie dla nas podstawą dla reprezentatywnego, choć niewyczerpującego przeglądu najważniejszych zagrożeń i metod przeciwdziałania im.

Przed rozpoczęciem omawiania poszczególnych zagrożeń podkreślmy, że właściwa aplikacji WWW konieczność otwarcia oprogramowania na parametry i żądania przesyłane z zewnętrznego świata plus bogaty zestaw technologii i formatów danych współdziałających w ramach aplikacji, czynią problem bezpieczeństwa tego typu oprogramowania szczególnie trudnym.

Trudność w zidentyfikowaniu podatności[2] naszej aplikacji na określonego rodzaju atak wiąże się z tym, że autor aplikacji myśli w terminach funkcjonalności, którą potrzebuje zrealizować, co samo w sobie bywa trudne i absorbujące. Tymczasem potencjalny intruz spojrzy na działający system nie przez pryzmat jego typowych przypadków użycia, ale jako na zestaw współdziałających ze sobą technologii, protokołów i formatów, oraz na specyficzne efekty przetwarzania nietypowych zestawów parametrów skierowanych do aplikacji. Innymi słowy, problemem jest to, że zagrożeń trzeba szukać częstokroć pośród tych aspektów, ze zgłębiania których mają nas zwalniać nowoczesne narzędzia, biblioteki i środowiska zorientowane na podnoszenie poziomu abstrakcji w pracy nad oprogramowaniem. Na szczęście jednak, podobnie jak gotowe zręby funkcjonalności dostarczane są przez ramę, tak też może ona zapewnić szereg mechanizmów chroniących przed typowymi rodzajami zagrożeń. Niemniej jednak, programista aplikacji powinien pozostać świadomym celu i sposobu działania takich zabezpieczeń. Wreszcie, niezależnie od skuteczności platformy, elementem przesądzającym o bezpieczeństwie aplikacji są właściwe decyzje projektanta i programisty aplikacji[3]. Wszak jeśli dajemy programiście do dyspozycji odpowiednie zasoby, to pozostaje ryzyko, że świadomie lub nie uczyni je dostępnymi dla intruza (to ten fakt właśnie sprawia, że konfiguracje dostępnych funkcjonalności na platformach providerów oferujących wsparcie wielu domen w ramach tego samego środowiska serwerowego bywają znacznie okrojone).

Podtrzymywanie stanu interakcji w aplikacji WWW

Protokół HTTP, stanowiący podstawę funkcjonowania systemu WWW, został pierwotnie, m.in. z myślą o serwowaniu czysto statycznych treści, zaprojektowany jako bezstanowy i bezpołączeniowy. W module poświęconym protokołom wspomnieliśmy, że wraz z rozwojem technologii WWW usunięto te ograniczenia: możliwość uzgodnienia pozostawienia połączenia otwartym pozwala sprawniej pobierać liczne powiązane ze sobą zasoby składające się na dokument WWW; z kolei mechanizm podtrzymania stanu w protokole HTTP (oparty na tzw. ciasteczkach (*cookies*)), pozwala na podtrzymanie informacji na czas dłuższy niż obsługa pojedynczego żądania.

---

[2] Będziemy tu kilkukrotnie używać słowa „podatność” jako odpowiednika anglojęzycznego *vulnerability*, rozumiejąc przez to brak odporności strony czy witryny na określonego rodzaju ataki.

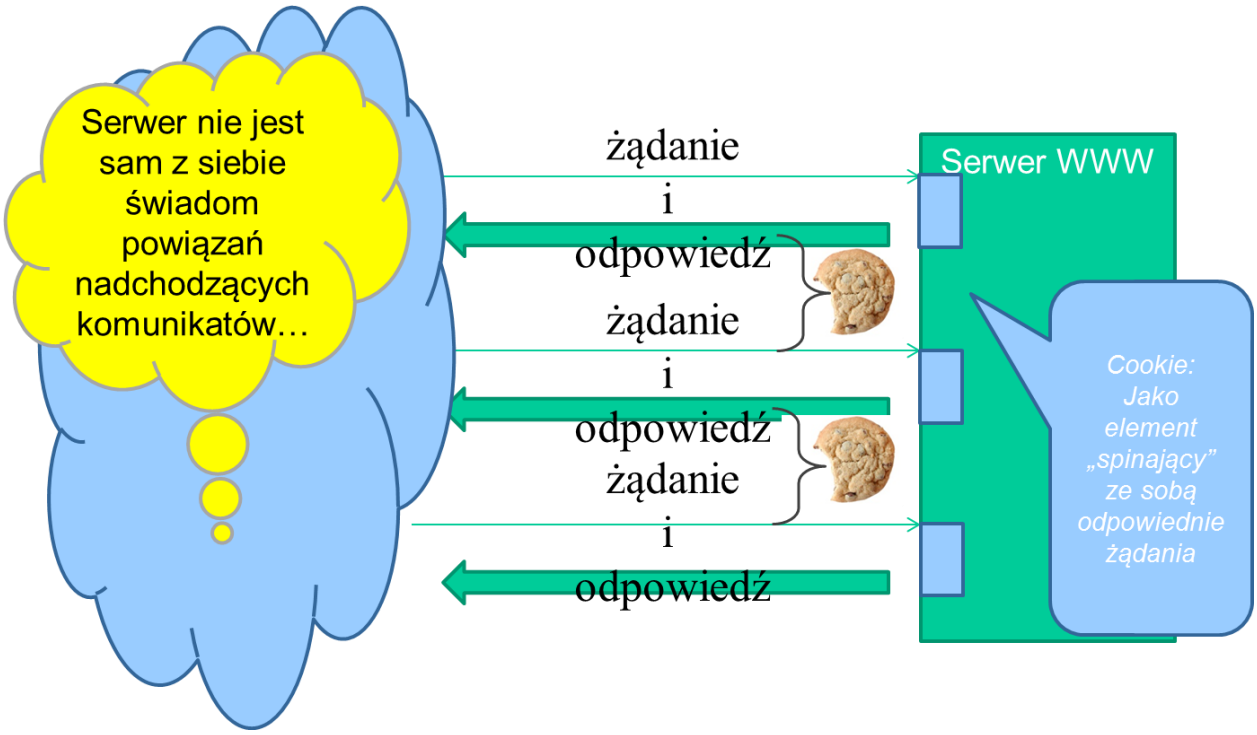
[3] Można ich rolę w zakresie bezpieczeństwa przyrównać do roli kierowcy samochodu: tam niezależnie od właściwości jezdnych i rozlicznych systemów bezpieczeństwa, niewłaściwe decyzje związane z prowadzeniem pojazdu mogą doprowadzić do katastrofy.

W dotychczasowych ćwiczeniach nie wykorzystywaliśmy explicite mechanizmu cookies. Ze względu na jego istotność tak dla funkcjonalności aplikacji jak i dla zagadnień bezpieczeństwa – warto go więc w tym miejscu przedstawić.

Kod aplikacji WWW działający po stronie serwera ma do dyspozycji wszelkie zasoby zlokalizowane w tym środowisku (w tym zwykle bazę danych), jak i szczegóły pochodzące z komunikatu żądania. Bez dostarczenia dodatkowego mechanizmu przekazywania danych – nie byłby jednak w stanie skorelować ze sobą kolejnych żądań kierowanych doń przez tego samego użytkownika. Przykładowo, w przypadku sklepu internetowego, niemożliwe byłoby sformułowanie polecenia „złóż zamówienie", skoro nie byłoby wiadomo, z jakimi wcześniejszymi poleceniami „włóż towar do koszyka" jest ono związane. Tej swoistej „amnezji" serwera WWW zapobiega mechanizm cookies. Termin „ciasteczko" jest w obszarze informatyki rozumiane jako „nieprzejrzysty element treści przechowywany przez pośrednika". W naszym przypadku tym pośrednikiem jest przeglądarka, zaś nieprzejrzystość wiąże się z tym, że treść zawarta w ciasteczku nie musi być rozumiana przez przeglądarkę.

Mechanizm cookies ilustruje poniższy rysunek. Trzy pokazane tutaj żądania pochodzą kolejno od tego samego użytkownika, z tego samego okna przeglądarki, jednakże, bez dodatkowego mechanizmu, po stronie serwera ta wiedza nie byłaby dostępna. Stąd zastosowanie cookie:

- Funkcjonalność działająca po stronie serwera ma zapotrzebowanie na zapamiętanie pewnych danych na temat współdziałania z danym klientem;
- W ramach obsługi bieżącego żądania, przygotowuje niezbędne do zapamiętania dane i przesyła, wraz z odpowiedzią serwera, w ramach nagłówków protokołu HTTP jako tzw. cookie.
- Przeglądarka zapamiętuje cookie w postaci małego pliku tekstowego. Z jej punktu widzenia istotne są metadane: jak długo ciasteczko pozostaje ważne oraz przy wywoływaniu jakich adresów URL (zob. dalej) należy je wysyłać w ramach kierowanego na serwer żądania. Te informacje możemy określić jako metadane ciasteczka. Sama treść zaś ciasteczka (dane) nie muszą być rozumiane przez przeglądarkę.
- Przy kolejnym żądaniu kierowanym do tego samego serwera, przeglądarka dołącza ciasteczko.
- Żądanie dociera na serwer jako potencjalnie należące do zupełnie innej interakcji z innym użytkownikiem, jednak dzięki zawartości przesłanego cookie interakcja zostaje zidentyfikowana i niezbędne dane zostają odtworzone na podstawie treści cookie.



Kontrola bieżąca	( )	*komentarz odpowiedzi*
Treść ciasteczka jest przekazywana pomiędzy serwerem a przeglądarką:		
wewnątrz części nagłówkowej dokumentu HTML		Nie.
jako ciało komunikatu o metodzie POST		Nie. Dostęp do nich jest potrzebny także przy żądaniach pozostałych metod.
poza protokołem HTTP		Nie. Ciasteczka wkomponowano w protokół HTTP
jako jeden z nagłówków HTTP	x	Tak.

Zauważmy więc, że w tym typowym scenariuszu wszelki zapis i odczyt danych w cookie odbywa się wyłącznie po stronie serwera.

Nie chcąc powielać tutaj wszystkich szczegółów technicznych dostępnych w specyfikacji standardu, ograniczmy się tutaj do następującej charakterystyki metadanych ciasteczka:

- Właściwość `expires` – podaje datę i czas uniwersalny wyznaczające termin obowiązywania ciasteczka. Po upływie tego terminu ciasteczko nie będzie stosowane. Przesłanie przez serwer ciasteczka o zadanej nazwie i z datą wcześniejszą niż obecna oznacza z kolei dyspozycję usunięcia ciasteczka z repozytorium przeglądarki. Brak właściwości `expires` cechuje natomiast tzw. ciasteczko sesyjne. Pozostanie ono w pamięci przeglądarki do momentu zamknięcia bieżącego okna.
- Właściwości `path` oraz `serwer` – wskazują, przy wywoływaniu jakich adresów URL dane ciasteczko należy przesyłać w żądaniu. Gdy właściwości te nie wystąpią, to domyślny zasięg obowiązywania ciasteczka jest najwęższy: będzie wysyłane wyłącznie przy wołaniu adresów URL zgodnych względem ścieżki wirtualnej (do ostatniego ‘/’) z adresem strony, która spowodowała założenie danego cookie. A więc np. http://www.example.org/kontakt/form.html oraz http://www.example.org/kontakt/index.html , ale już nie

http://www.example.org/oferta/index.html . Dlatego też często spotykamy właściwość `path` z wartością równą „/”, co oznacza efektywnie wszystkie strony danej witryny. Podobnie, brak właściwości serwer będzie oznaczał, że będzie ono dotyczyło tylko adresów wskazujących na tę samą witrynę (czyli np. http://www.example.com), zaś jej użycie pozwala poszerzyć zakres obowiązywania ciasteczka na różne witryny, ale w ramach jednej wykupionej domeny (a więc np. wszystkie serwery i pod-domeny w domenie `example.org` albo wszystkie serwery i pod-domeny `example.org.pl`).

- Flaga `secure` – instruuje przeglądarkę, że ciasteczko (naturalnie po spełnieniu zarazem ww. kryteriów) ma być wysyłane w żądaniach kierowanych do serwera tylko wówczas, gdy zestawione jest połączenie bezpieczne (`https`).
- Flaga `HttpOnly` – stanowi dyspozycję, by ciasteczko nie było po stronie przeglądarki dostępne do manipulowania (odczyt / modyfikacja) przez JavaScript.

Z powyższej charakterystyki widzimy, że cookies stanowią mechanizm zapisu danych przeznaczony dla kodu działającego po stronie serwera, chociaż (podobnie jak w przypadku innych aspektów funkcjonalności aplikacji) może być (o ile nie wygaszono – j.w. tej opcji) wykorzystywany również po stronie przeglądarki. Związane z bieżącym dokumentem ciasteczka mogą być (wraz ze swoimi metadanymi) odczytywane i modyfikowane z poziomu JS poprzez właściwość `window.cookie` .

<<ANIMACJA A2 – Obsługa i wykorzystania cookies po stronie serwera i po stronie klienta.>>

Gwoli kompletności warto też wspomnieć o rozwiązaniu, wprowadzonym stosunkowo niedawno w API przeglądarek w ramach szeroko rozumianej rodziny technologii HTML5, a służącemu z kolei utrwalaniu danych wyłącznie na potrzeby warstwy klienta. Nosi ono nazwę Local Storage (dostępne w JS poprzez właściwość `window.localStorage`) i pozwala gromadzić po stronie przeglądarki trwałe dane związane z adresem domenowym danego dokumentu. Proszę zapoznać się z jego działaniem – np. korzystając z (można równocześnie dokonać poglądu środowiska działania skryptu, np. korzystając ze wspomnianego wcześniej debuggera Firebug). Jego przewagą nad tradycyjnymi cookies jest brak konieczności przesyłania danych między klientem a serwerem (skoro potrzebne są tylko po stronie klienta) oraz mniej restrykcyjne ograniczenia na objętość tak przechowywanych danych.

Podobnie, interfejs programistyczny JS obejmuje właściwość `window.sessionStorage` . Jak wskazuje nazwa, w tym przypadku mamy do czynienia z danymi utrwalanymi tylko na czas funkcjonowania danego okna przeglądarki.

Wreszcie – należy wspomnieć o wspieranym przez praktycznie wszystkie technologie dokumentów dynamicznych WWW – pojęciu zmiennych sesyjnych. Są to zmienne, które programista może wykorzystywać do przechowywania danych związanych ze współpracą z danym użytkownikiem (np. lista towarów umieszczonych w koszyku, fakt zalogowania, ustawienia widoku strony itp.). Stanowią one warstwę abstrakcji nad faktycznym mechanizmem podtrzymywania tego rodzaju danych. Oznacza to, że sposób zapewnienia tego efektu może być różny. Np. wszystkie dane sesyjne mogą być umieszczone w cookie. Takie rozwiązanie nie angażuje wprawdzie zasobów (pamięć) po stronie serwera, ma jednak inne wady:

- Ze względów wydajności komunikacji (a wręcz – ograniczeń technicznych na mechanizm cookies) – objętość tych danych jest ograniczona.
- Przekazywanie takich danych (zarówno zapisywanych jak i odczytywanych wyłącznie przez funkcjonalność działającą na serwerze) na stronę klienta jest niezasadne.
- Dodatkowo – przesyłanie niektórych danych może okazać się problematyczne z punktu widzenia bezpieczeństwa (dane wrażliwe).

Z powyższych względów zmienne sesyjne są zwykle implementowane w oparciu na zasobach serwera: dane są gromadzone na serwerze, zaś ciasteczko służy podtrzymaniu jedynie unikalnego identyfikatora sesji, który służy funkcjonalności serwerowej do wyłuskania odpowiedniego zestawu danych. Tak rozumiany identyfikator sesji będzie rozpatrywany niżej przy omówieniu głównych zagrożeń dla bezpieczeństwa aplikacji. W szczególności, zmienna sesyjna może zawierać informację, że użytkownik dokonał pomyślnego uwierzytelnienia. Tym samym dysponowanie wartością identyfikatora takiej sesji jest w krótkim horyzoncie czasowym niemal równoważne posiadaniu danych dostępowych do konta użytkownika.

Kontrola bieżąca	( )	*komentarz odpowiedzi*
Podtrzymanie stanu interakcji klienta z serwerem w taki sposób, że dane pozostają niedostępne dla skryptów działających po stronie przeglądarki jest cechą mechanizmu:		
cookies z opcją `HttpOnly`	x	Tak.
HTML5 Local Storage		Nie.
HTML5 Session Storage		Nie.
parametrów przekazywanych w ramach *query string*		Nie.

Ranking OWASP – omówienie

W selekcji najistotniejszych zagrożeń bezpieczeństwa aplikacji WWW bardzo pragmatyczne podejście przyjmuje OWASP w swoim rankingu 10 czołowych zagrożeń. Uwzględnia on bowiem nie samą popularność danego problemu, lecz zamiast tego – waży takie czynniki jak: rozpowszechnienie danej podatności na atak, łatwość jej odkrycia, łatwość jej wykorzystania oraz dotkliwość skutków ewentualnego ataku.

Z powyższych względów dokonamy tu krótkiego przeglądu i komentarza głównych zagrożeń z rankingu OWASP 2013 Top 10.

Injections

Najszerzej chyba znaną, a zarazem najdonioślejszą według ww. rankingu jest kategoria ataków zwana *injection*. Nazwa pochodzi od „wstrzykiwania" niepożądanych danych lub komend poprzez różnorakie parametry przyjmowane przez aplikację ww. oraz następującego w jego konsekwencji niepożądanego (i częstokroć nieprzewidzianego przez autora aplikacji) zachowania aplikacji.

Jednym z częściej występujących wariantów jest tzw. *SQL Injection*, czyli przekazanie do aplikacji takiego parametru, który w zamyśle autora aplikacji miał stać się częścią wyrażenia SQL, zaś wskutek odpowiedniego spreparowania – wypacza działanie danego wyrażenia i opartych na jego rezultacie dalszych kroków przetwarzania, bądź też „przemyca" wraz z parametrem dodatkową instrukcję SQL. Ten drugi przypadek, w nieco wyjaskrawionej formie, przedstawiono w następującym komiksie: https://xkcd.com/327/ .

W przykładzie tym intruz mógł posłużyć się wręcz istniejącym interfejsem formularza WWW, umieszczając w nim odpowiednio rozbudowaną zawartość. Przewidując, że przekazany parametr zostanie włączony (być może – bez należytego przefiltrowania) do zapytania SQL, zawarł w nim znaki zamykający wartość tekstową, zamykający nawias należący do podwyrażenia w klauzuli WHERE, średnik zamykający wyrażenie, kolejną instrukcję SQL (stanowiącą właściwy atak) oraz symbol komentarza zapobiegający przetworzeniu reszty wyrażenia przygotowanego przez autora aplikacji. W tym przypadku efektem jest usunięcie danych z bazy. Naturalnie, w analogiczny sposób przeprowadzić można subtelniejsze ataki – np., zamiast dodania kolejnej instrukcji – polegające np. na

„rozmiękczeniu” predykatu w wyrażeniu selekcji obsługującym uwierzytelnianie tak, by zapytanie skierowane do bazy zwróciło niepusty rezultat nawet w przypadku braku poprawnie podanego hasła.

Należy podkreślić, że w analogiczny sposób przekazane do aplikacji mogą być inne newralgiczne dla przetwarzania dane – np. w postaci instrukcji stosowanego po stronie serwera języka programowania, instrukcji funkcjonującego tam interfejsu programistycznego, instrukcji systemu operacyjnego czy też – kodu JS w strukturze strony HTML generowanej celem zaserwowania jej klientowi.

(dość obszerną ilustrację SQL injection znajdziemy m.in. w następującym materiale: http://www-07.ibm.com/events/au/hacking/i/AppScan_POT_v02.pdf (sekcja ”SQL Injection”).

Główne zalecenie w zapobieganiu tego typu zagrożeniom zawarto nawet w morale powyższej historyjki: dane przekazywane do aplikacji muszą podlegać bardzo starannemu przefiltrowaniu. Mamy tu na myśli ich walidację po stronie serwera tuż przed ich faktycznym użyciem. Proszę bowiem zwrócić uwagę, że np. zablokowanie przez autora możliwości wpisania niektórych znaków w interfejsie użytkownika (tj. tutaj – zapewne w rubryce formularza WWW) nie powstrzyma intruza przez manualnym spreparowaniem treści żądania (umożliwiające to narzędzia były już wspomniane w jednym z poprzednich modułów).

Stosowanie ram programistycznych takich jak Express i inne współdziałające z nim pakiety Node.js ułatwia tego rodzaju filtrowanie, zaś przygotowanie parametryzowanego wyrażenia SQL i dołączanie doń odpowiednich parametrów ograniczają ryzyko wypaczenia jego działania, które jest szczególnie wysokie w przypadku prostej konkatenacji łańcuchów tekstu.

Wadliwe uwierzytelnianie i zarządzanie sesją

Powyższa kategoria jest zbiorcza – obejmuje różnorodne podatności aplikacji, grożące przede wszystkim możliwością wycieku danych, kradzieży tożsamości czy też nieautoryzowanych zmian stanu aplikacji.

Do zagrożeń tej kategorii zaliczamy również gromadzenie po stronie serwera danych uwierzytelniających użytkowników w sposób niezabezpieczony (a więc – podatny na wyciek lub utratę wskutek włamania).

Większość zagrożeń tej kategorii wiąże się natomiast z bezpieczeństwem identyfikatora sesji. I tak, niezalecane jest umieszczanie go jako parametru w adresie URL (możliwość wycieku przy przekazaniu linku lub większa łatwość przechwycenia w komunikacji), jak też przekazywanie danych uwierzytelniających, ale też samych identyfikatorów sesji przez niezabezpieczone łącze.

Zagrożenie stanowią identyfikatory sesji niezmieniające się przez dłuższy czas oraz zależne od parametrów pochodzących od danego użytkownika – a więc łatwiejsze w odgadnięciu.

Wreszcie – w tej kategorii zagrożeń znajduje również podatność na ataki zwane *session fixation*. W tym przypadku istnieje możliwość podszycia się pod danego użytkownika bez konieczności odgadnięcia czy wykradnięcia jego identyfikatora sesji. Intruz podsuwa atakowanemu użytkownikowi hiperłącze czy formularz powodujące, że ten będzie uwierzytelniał się (logował) w pewnym newralgicznym serwisie podając identyfikator sesji, który wytworzył (rozpoczynając swoją własną procedurę logowania w tym serwisie – zob. ilustrację w http://guides.rubyonrails.org/security.html#session-fixation). Gdy dojdzie do uwierzytelnienia, intruz będzie mógł podszyć się pod użytkownika posługując się owym numerem sesji. Metodą zapobieżenia temu zagrożeniu jest wymuszanie zmiany identyfikatora sesji po zalogowaniu.

Używamy tutaj sformułowań "atak na aplikację" oraz "atak na użytkownika". Trzeba podkreślić, że w przypadku niektórych rodzajów zagrożeń (a należy do nich m.in. omówiony niżej XSS) oba te podmioty są przedmiotem ataku. Aplikacja - w sensie wykorzystania jej podatności na wplecenie weń wrogiej treści, zaś użytkownik - w sensie dążenia do wykradnięcia jego danych czy tożsamości.

W przypadku zagrożeń, w których mowa wyłącznie o ataku na aplikację (np. SQL injection zorientowany na przełamanie zabezpieczeń dostępu) - często postrzegamy aplikację serwerową jako główną linię obrony, zaś przeglądarkę (lub inny program kliencki) - jako znajdujący się potencjalnie we władaniu mającego złe zamiary intruza. Nieco inaczej jest w przypadku XSS. Tutaj potencjalnym sojusznikiem zagrożonego atakiem użytkownika jest jego przeglądarka wraz z zaimplementowanymi w niej politykami bezpieczeństwa.

Kontrola bieżąca	( )	*komentarz odpowiedzi*
Źródłem zagrożenia typu SQL Injection jest przede wszystkim:		
łatwe do odgadnięcia hasło do bazy danych		Nie.
nieskuteczne filtrowanie danych wejściowych przez funkcjonalność aplikacji WWW	x	Tak.
wadliwa implementacja strony logowania		Nie. Zagrożenia SQL injection mogą wystąpić również na innych stronach.
luka bezpieczeństwa w implementacji serwera WWW		Nie. Autor aplikacji może sprowokować to zagrożenie nawet wykorzystując w pełni pozbawione podatności oprogramowanie serwera.

XSS – Cross Site Scripting

Zagrożenie to (opatrzone akronimem XSS a nie CSS, by zapobiec konfliktowi nazw z kaskadowymi arkuszami stylów) należy do najbardziej znanych. W rankingu OWASP w roku 2007 zajmowało nawet pierwsze miejsce.

Źródłem problemu przy atakach XSS jest niedostateczna kontrola przez aplikację przekazywanych doń parametrów, które stają się następnie częścią generowanej dla klienta strony. Wyróżniamy tutaj ataki XSS tzw. **składowane**, w których zmanipulowana zawartość zostaje zapisana po stronie serwera i jest dostępna dla wszystkich odwiedzających stronę (np. w postaci wpisu na forum), jak również tzw. ataki **odbite**. Te drugie są bardziej wyrafinowane, gdyż polegają na spreparowaniu parametrów (np. w hiperłączu) podsuwanych użytkownikowi i nie pozostawiają wobec tego równie wyraźnego trwałego śladu na zaatakowanym serwerze. Przebieg ataku XSS ilustruje np. niniejszy schemat: http://excess-xss.com/#reflected-xss

Jak widzimy, przedmiotem ataku są tu raczej zasoby zlokalizowane po stronie klienta. Wiemy już jednak, że i tutaj mogą wystąpić bardzo poważne konsekwencje – uzyskanie identyfikatora cookie może doprowadzić wręcz do kradzieży tożsamości użytkownika.

Warto w tym kontekście wspomnieć o tzw. polityce tego samego źródła pochodzenia (same-origin policy). Jest ona realizowana w podobny co do zasady sposób przez wszystkie przeglądarki, choć nie jest w pełni ustandaryzowana. Sprowadza się w praktyce do zabronienia odczytu pobranego z witryny B zasobu, jeśli żądanie wykonuje skrypt pochodzący z witryny A. Źródło pochodzenia jest w tym przypadku określone jak złożenie 'schematu URI' (np. `http` lub `https`), nazwy domenowej hosta oraz numeru portu. Reguła ta jednak nie zapobiegnie interakcji, która następuje w ww. nakreślonym scenariuszu ataku XSS. Żądanie GET zawierające wykradzione dane wciąż może być wysłane do obcej witryny. Funkcjonalność taka (wysoce użyteczna np. przy pobieraniu obrazów czy arkuszy stylów) jest

dostępna w funkcjonujących od lat rozwiązaniach, toteż wprowadzanie w nowszych technologiach (jak obiekt `XmlHttpRequest` w technologii AJaX) silniejszych obostrzeń byłoby tutaj niezasadne i nieefektywne.

Przy okazji same-origin policy często przytacza się specyfikację Cross-Origin Resource Sharing (CORS). Dopuszcza ona czytanie przez skrypt zasobów z innej domeny, jednakże pod warunkiem spełnienia (sprawdzanego przez serwer obsługujący żądanie) zasobu kryterium: skierowane żądanie HTTP musi posiadać nagłówek Origin, wskazujący pochodzenie skryptu zwracającego się z żądaniem. Serwer ów deklaruje (i udostępnia do odpytania) listę witryn, żądania od których respektuje. Tylko przy spełnieniu takich warunków zgodności żądanie "obcego" skryptu zostanie zrealizowane.

Zalecenia związane z zapobieganiem tego typu zagrożeniom są podobne jak w przypadku *injection*: niezbędne jest odpowiednie kontrolowanie i filtrowanie parametrów wpływających na treść generowaną dla użytkownika.

Niezabezpieczone bezpośrednie odwołania do obiektów

Zagrożenie to występuje w sytuacji, gdy istnieje łatwa do odkrycia zależność pomiędzy parametrami przekazywanymi do aplikacji (np. w postaci łańcucha URL) a pobieranym lub modyfikowanym wskutek takiego żądania zasobem. Intruz może odgadnąć nazwy innych interesujących go zasobów (wierszy w bazie danych, plików, katalogów itp.) i spreparować odpowiednie żądanie. Stąd też fakt, że danemu użytkownikowi nie zaprezentowano na liście dostępnych dlań opcji danego parametru nie stanowi dostatecznego zabezpieczenia. Niniejszy materiał http://www-07.ibm.com/events/au/hacking/i/AppScan_POT_v02.pdf (w sekcji ”4 - Insecure Direct Object Reference”) zawiera ilustrację takiego ataku. Tym ciekawszą, że obejmuje ona przykład nieskutecznego odfiltrowywania niedozwolonego rozszerzenia nazwy pliku.

Niewłaściwa konfiguracja zabezpieczeń

Ta kategoria zagrożeń, zgodnie z nazwą, leży raczej w sferze konfiguracji niż programowania. Obejmuje różne niepożądane ustawienia konfiguracyjne – począwszy od najbardziej jaskrawych (np. pozostawienie aktywnej opcji zdalnego logowania z domyślnymi danymi uwierzytelnienia), poprzez zwiększające niepotrzebnie podatność na ataki (włączone mechanizmy, które faktycznie nie są wykorzystywane przez naszą aplikację), skończywszy na ustawieniach eksponujących szczegóły implementacyjne aplikacji (np. możliwość listowania zawartości katalogów zasobów WWW czy też komunikaty błędów listujące fragmenty kodu źródłowego aplikacji).

Ekspozycja danych wrażliwych

Jest to kategoria zagrożeń podobna do powyższej. Obejmuje takie podatności jak gromadzenie i przesyłanie danych wrażliwych w postaci niezabezpieczonej kryptograficznie, czy też błędne lub zbyt słabe konfiguracje mechanizmów kryptografii.

Brak funkcjonalnej kontroli dostępu

Zagrożenia z tej kategorii są związane z błędem projektowym polegającym na tym, że kierowane do serwera żądania dotyczące newralgicznych danych czy funkcjonalności nie podlegają autoryzacji, bądź mechanizmy autoryzacji są nieskuteczne. Przykładowo, niedopuszczalna jest sytuacja, w której funkcjonalność adresowana do grona osób uprawnionych jest chroniona przez pozostałymi użytkownikami jedynie poprzez nieprezentowanie im adresu URL pod którym została udostępniona. Z efektywnym brakiem kontroli dostępu mamy do czynienia również wtedy, gdy przy uruchamianiu danej funkcjonalności sprawdzane są jedynie łatwe do odgadnięcia dane przekazane wcześniej przez serwer i znajdujące się w przesłanym żądaniu (np. flaga w rodzaju `isAdmin=true`). Rola, jaką pełni w aplikacji wywołujący żądanie użytkownik musi być zweryfikowana przy każdym wywołaniu.

Cross Site Request Forgery (CSRF)

Ten rodzaj ataku stanowi przykład, że ułatwienia oferowane użytkownikom aplikacji mogą prowadzić do obniżenia bezpieczeństwa. CSRF jest w pewnym sensie odwróceniem ataku XSS. Otóż w przypadku XSS mieliśmy do czynienia z przekazaniem danych związanych z zaatakowaną stroną do strony posiadanej przez intruza. Tu zaś (zob. http://guides.rubyonrails.org/security.html#cross-site-request-forgery-csrf) – na stronie znajdującej się we władaniu intruza znajduje się odsyłacz powodujący wywołanie strony atakowanego serwisu. W powyższym przykładzie wywołanie, po odwiedzeniu owej niebezpiecznej strony, następuje samoczynnie i być może wręcz w sposób niezauważony dla użytkownika – jako próba pobrania zasobu opisanego jako obraz. Ponieważ zaatakowana aplikacja (tutaj – www. webapp.com) oferowała użytkownikowi możliwość zapamiętania jego zalogowania, to wykonana na skutek ataku akcja usunięcia zasobu mogła się odbyć w sposób niezauważony, bez uwierzytelniania. Inne nazwy tego typu ataków to m.in. *session riding* lub *one-click attack*. Eksponują one fakt, że atak bazuje na istniejącym zestawieniu sesji z danym serwisem i uwierzytelnienia w jego ramach.

<<ANIMACJA A3 – Atak typu CSRF>>

Wywołania newralgicznych funkcji naszej aplikacji możemy zabezpieczać przed zagrożeniem CSRF wymuszając ponowne uwierzytelnienie przez użytkownika (spotykane np. przy wołaniu opcji modyfikacji danych kontaktowych użytkownika). W pozostałych funkcjonalnościach zabezpieczenie może stanowić mechanizm zamieszczania w żądaniu uprzednio udostępnionego przez serwer żetonu bezpieczeństwa. Zob. np. pakiet **csurf** dla Express.

Stosowanie komponentów o znanych podatnościach

Zapewnienie bezpieczeństwa aplikacji w praktyce wiąże się z koniecznością regularnych aktualizacji oprogramowania. W wielu technologiach rozwijanych na zasadach open-source, informacja o wykrytych podatnościach jest szybko upowszechniana. Pozwala to na sprawne przygotowanie poprawek, jednak do momentu ich instalacji w środowisku danej aplikacji, powszechna wiedza o danej podatności zwiększa ryzyko ataku. Również w przypadku poprawek oprogramowania o zamkniętym kodzie, istnieje ryzyko, że intruz dokona rozbioru opublikowanych dlań łat i zidentyfikuje podatność. Zatem również i w tym scenariuszu sprawne aktualizacje mogą okazać się niezmiernie ważne.

Kontrola bieżąca	( )	*komentarz odpowiedzi*
Gdy aplikacja żąda ponownego uwierzytelnienia się zalogowanego użytkownika przy wywoływaniu przezeń szczególnie newralgicznej funkcjonalności, to należy się domyślać, że:		
użytkownik padł ofiarą ataku zmierzającego do wyłudzenia hasła		Nie.
aplikacja realizuje zabezpieczenie przed zagrożeniami SQL injection		Nie.
aplikacja realizuje zabezpieczenie przed zagrożeniami CSRF	x	Tak.
aplikacja wykazuje podatność „brak funkcjonalnej kontroli dostępu”		Nie.

Nieskontrolowane przekierowania i przekazania

Zarówno przekierowanie (redirect) jak i przekazanie (forward) prowadzą do uruchomienia wywołania na serwerze zasobu innego niż pierwotnie wywołany. W przypadku *redirect* (znanego nam choćby ze wzorca projektowego PRG), interakcja obejmuje przeglądarkę (odpowiedni nagłówek instruuje ją do wykonania ponownego żądania skierowanego pod inny adres URL), zaś dla *forward* – jest realizowana

w całości po stronie serwera. Ta, być może subtelna, różnica implikuje różne rodzaje zagrożeń związanych z niedostatecznym skontrolowaniem docelowych adresów. W przypadku *forward* istotnym zagrożeniem jest możliwość obejścia za jego pomocą mechanizmów funkcjonalnej kontroli dostępu, które zafunkcjonowałyby, gdyby adres docelowy był wołany wprost. W przypadku *redirect* z kolei – zagrożenia mogą obejmować np. *phishing* – przekierowanie użytkownika (z podatnej na ten rodzaj ataku strony zaufanej witryny) do podstawionej przez intruza strony przypominającej tę oryginalną witrynę i wyłudzenie danych.

Podsumowanie

Dokonany powyżej przegląd zagrożeń można zsyntetyzować do kilku zasadniczych postulatów bezpieczeństwa:

- Staranna kontrola wszystkich nadsyłanych parametrów,
- Unikanie zbędnego przesyłania danych,
- Ograniczenie dostępności zbędnych opcji i uprawnień,
- Kontrolowanie parametrów i uprawnień tak, jak gdyby nie istniała żadna dbająca o nie funkcjonalność po stronie klienta,
- Aktualizacja oprogramowania,
- Znajomość mechanizmów zagrożeń i narzędzi służących przeciwdziałaniu im,
- Staranna kontrola wszystkich nadsyłanych parametrów[4].

Mnogość kanałów przekazywania oraz formatów treści czyni problematycznym filtrowanie negatywne, rozumiane jako usuwanie wszystkich niepożądanych czy podejrzanych elementów. Z tego też względu, zamiast eliminowania tychże (zasada czarnej listy) lepiej tam, gdzie to możliwe, przyjąć tzw. pozytywne podejście do filtrowania (biała lista). Oznacza to, że z założenia pomijane są wszystkie treści z wyjątkiem zestawu wartości oczekiwanych.

Znacznym wsparciem dla programisty aplikacji WWW dbającego o bezpieczeństwo są odpowiednio dojrzałe mechanizmy platformy / ramy programistycznej. Podnosząc poziom abstrakcji typowych zadań dostarczają wraz z nim odpowiednich mechanizmów zabezpieczeń. Niemniej jednak, świadomość rodzajów zagrożeń ogólnych oraz specyficznych dla danej technologii są również i w tym przypadku niezbędne by skutecznie posłużyć się tym narzędziem. O ile dokonany tu przegląd ma charakter dość podstawowy, to zachęcam do bliższego zapoznania się ze specyficznymi dla Node.js wskazówkami nt. bezpieczeństwa, dostępnymi np. na https://medium.com/@tkssharma/secure-node-js-apps-7613973b6971 .

[4] Powtórzenie jest zamierzone. Ten postulat ma kluczowe znaczenie.

Zadanie

Ze względu na wprowadzający charakter materiału w tym module, nie zamierzamy tutaj zagłębiać się w zaawansowane aspekty zabezpieczeń aplikacji opartej na Node.js. Zamiast tego, wykorzystajmy niniejsze zadanie do zapoznania się z rozwiązaniami podtrzymania stanu interakcji klienta z aplikacją.

Korzystając z dotychczas zdobytych umiejętności budowania formularzy i kontrolerów w Node.js/Express, proszę sporządzić formularz, który umożliwi przesłanie kilku parametrów (np. wartości tekstowych), a następnie ich przetworzenie tak, aby każdy z atrybutów został utrwalony w odmienny sposób. Łącznie powinny zostać zademonstrowane:

- zapisanie wartości w ciasteczku terminowym;
- zapisanie wartości w ciasteczku sesyjnym;
- zapisanie wartości w zmiennej sesyjnej;
- zapisanie wartości w `localStorage`;

Proszę również sporządzić widok, który będzie wypisywał te wartości na stronie.

Pomocna może w tym być część materiałów z modułu poprzedzającego.