Testowanie bezpieczeństwa aplikacji internetowych. Receptury

Autor: Paco Hope, Ben Walther
Tłumaczenie: Radosław Meryk
ISBN: 978-83-246-2208-5
Tytuł oryginału: Web Security Testing Cookbook
Format: 168x237, stron: 312

Poznaj i wykorzystaj mechanizmy testowania zabezpieczeń, a nikt nie prześlizgnie się przez Twoją witrynę!

- Jak zainstalować i skonfigurować narzędzia do testowania zabezpieczeń?
- Jak szybko i sprawnie znaleźć problemy w aplikacjach?
- Jak wykorzystywać testy powtarzalne?

Witryny internetowe oraz ich aplikacje stanowią swoistą wirtualną furtkę do wszystkich korporacji i instytucji. Jak zatem zadbać, aby nikt niepożądany nie przedostał się do środka? Co sprawia, że witryna jest naprawdę bezpieczna? I w jaki sposób testować aplikację, aby nie był to proces żmudny i czasochłonny, a raczej sprawny i skuteczny? Oto rozwiązanie – niniejsza książka zawiera proste receptury, dzięki którym z łatwością znajdziesz luki w aplikacjach, zanim zrobią to różni hakerzy.

Książka „Testowanie bezpieczeństwa aplikacji internetowych. Receptury" to napisany zrozumiałym językiem podręcznik, dzięki któremu szybko poznasz mechanizmy testowania zabezpieczeń. Praktyczne przykłady zawarte w tym przewodniku sprawią, że szybko nauczysz się włączać systemy zabezpieczeń do standardowych procedur kontroli aplikacji. Bez problemu stworzysz testy dotyczące funkcji AJAX, a także przeprowadzisz rozbudowane, wieloetapowe testy podatności na klasyczne problemy: skrypty krzyżowe oraz wstrzykiwanie kodu.

- Bezpieczeństwo oprogramowania
- Instalacja darmowych narzędzi i rozszerzeń
- Kodowanie danych w Internecie
- Manipulowanie danymi wejściowymi
- Fałszowanie informacji przesyłanych w nagłówkach przez przeglądarki
- Przesyłanie na serwer plików o dużej objętości
- Obchodzenie ograniczeń interfejsu użytkownika
- Autoryzacja masowego skanowania
- Ataki przeciwko aplikacjom AJAX
- Manipulowanie sesjami
- Testy wielostronne

Niech bezpieczeństwo Twoich aplikacji nie spędza Ci snu z powiek!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 4.

Kodowanie danych w internecie

Jeśli chodzi o obserwację,
los nagradza tylko przygotowane umysły.

— Louis Pasteur

Pomimo że aplikacje internetowe spełniają cały szereg różnych funkcji, mają różne wymagania i oczekiwane zachowania, istnieją podstawowe technologie i bloki budulcowe, które pojawiają się częściej niż inne. Jeśli zapoznamy się z tymi blokami budulcowymi i opanujemy je, będziemy dysponować uniwersalnymi narzędziami, które można zastosować do różnych aplikacji internetowych, niezależnie od specyficznego przeznaczenia aplikacji lub technologii użytych do ich zaimplementowania.

Jednym z takich podstawowych bloków budulcowych jest kodowanie danych. W aplikacjach internetowych pomiędzy serwerem WWW a przeglądarką dane przesyłane są na wiele sposobów. W zależności od typu danych, wymagań systemu oraz preferencji określonego programisty dane te mogą być zakodowane lub spakowane z wykorzystaniem wielu różnych formatów. W celu przygotowania użytecznych przypadków testowych często trzeba zdekodować dane, wykonać na nich operacje i ponownie je zakodować. W szczególnie skomplikowanych sytuacjach trzeba przeliczyć prawidłowe wartości testów integralności takie jak sumy kontrolne lub skróty (ang. *hash*). Znakomita większość testów w środowisku internetowym obejmuje manipulowanie parametrami przekazywanymi pomiędzy serwerem a przeglądarką. Zanim jednak przystąpimy do wykonywania operacji z parametrami, powinniśmy zrozumieć, w jaki sposób są one pakowane i przesyłane.

W niniejszym rozdziale opowiemy o rozpoznawaniu, dekodowaniu i kodowaniu różnych formatów: Base64, Base36, czasu Unix, kodowania URL, kodowania HTML i innych. Informacje zamieszczone w niniejszym rozdziale nie mają pełnić roli materiałów referencyjnych (istnieje wiele dobrych materiałów na ten temat). Mają one jedynie pomóc w rozpoznaniu podstawowych formatów i sposobów manipulowania nimi. Dopiero gdy będziemy mieli pewność, że aplikacja zinterpretuje dane wejściowe w sposób, jakiego się spodziewamy, będziemy mogli uważnie opracować testowe dane.

Typy parametrów, które będziemy analizować, są wykorzystywane w wielu niezależnych miejscach podczas interakcji z aplikacją internetową. Mogą to być ukryte wartości pól formularzy, parametry GET przekazywane za pośrednictwem adresów URL oraz wartości w obrębie plików

cookie. Mogą to być krótkie informacje, na przykład sześcioznakowy kod rabatu, lub rozbudowane dane, na przykład setki znaków o wewnętrznej wielowarstwowej strukturze. Tester powinien przeprowadzić testy przypadków granicznych oraz testy negatywne dotyczące *interesujących* przypadków. Nie można jednak stwierdzić, co jest interesujące, jeśli się nie rozumie formatu danych. Trudno jest metodycznie wygenerować wartości graniczne i dane testowe, jeśli nie zna się struktury danych wejściowych. Na przykład jeżeli zobaczymy ciąg `dGVzdHVzZXI6dGVzdHB3MTIz` w nagłówku HTTP, możemy czuć pokusę, aby zmodyfikować go w losowy sposób. Wystarczy jednak zdekodować ten ciąg za pomocą dekodera Base64, aby dowiedzieć się, że kryje się pod nim ciąg `testuser:testpw123`. W tym momencie Czytelnik powinien mieć znacznie lepsze rozeznanie na temat danych i wiedzieć, że należy je modyfikować zgodnie ze sposobem ich wykorzystania. Dzięki temu można przygotować prawidłowe przypadki testowe, które są właściwie ukierunkowane na działanie aplikacji.

4.1. Rozpoznawanie binarnych reprezentacji danych

Problem

Zdekodowaliśmy pewne dane w obrębie parametrów, pól wejściowych lub pliku danych i chcemy przygotować dla nich właściwe przypadki testowe. Powinniśmy określić, jakiego typu są to dane, abyśmy mogli przygotować dobre przypadki testowe pozwalające na manipulowanie danymi w interesujący sposób.

Analizie poddamy następujące rodzaje danych:

- szesnastkowe (Base16),
- ósemkowe (Base8),
- Base36.

Rozwiązanie

Dane szesnastkowe

W skład cyfr szesnastkowych (Base16) wchodzą znaki cyfr dziesiętnych 0 – 9 oraz litery A – F. Czasami są pisane samymi wielkimi bądź samymi małymi literami. Rzadko jednak można spotkać pisownię, w której wielkość tych liter jest mieszana. Występowanie dowolnych liter, które w alfabecie są za literą F, oznacza, że nie mamy do czynienia z danymi Base16.

Chociaż informacje, które tu przedstawiamy, to komputerowy elementarz, warto go powtórzyć w kontekście testowania. Każdy bajt danych jest reprezentowany w wyniku przez dwa znaki. Warto tu zwrócić uwagę na kilka szczególnych przypadków, na przykład że ciąg `00` oznacza bajt o wartości `0`, czyli `NULL`. Jest to jedna z naszych ulubionych wartości granicznych wykorzystywanych do testowania. Z kolei ciąg `FF` to `255` lub `-1` w zależności od tego, czy mamy do czynienia z wartością ze znakiem, czy bez. To kolejna nasza ulubiona wartość graniczna. Do innych interesujących wartości należy `20` — kod ASCII znaku spacji oraz `41` — kod ASCII wielkiej litery `A`. Powyżej kodu ASCII `7F` nie ma drukowalnych znaków. W większości języków programowania wartości szesnastkowe można rozróżnić po literach `0x` na początku. Jeśli zobaczymy ciąg `0x24`, powinniśmy instynktownie interpretować tę wartość jako

liczbę szesnastkową. Inny popularny sposób reprezentacji wartości szesnastkowych polega na oddzieleniu poszczególnych bajtów dwukropkami. W ten sposób często przedstawiane są sieciowe adresy MAC, wartości MIB protokołu SNMP, certyfikaty X.509, a także inne protokoły i struktury danych korzystające z kodowania ASN.1. Na przykład adres MAC można przedstawić w następujący sposób: `00:16:00:89:0a:cf`. Należy zwrócić uwagę na to, że niektórzy programiści pomijają niepotrzebne wiodące zera. Zgodnie z tym powyższy adres MAC można przedstawić w następujący sposób: `0:16:0:89:a:cf`. Chociaż w takim ciągu niektóre dane są pojedynczymi cyframi, nie oznacza to, że nie jest to seria bajtów szesnastkowych.

Dane ósemkowe

Kodowanie ósemkowe — Base8 — jest stosunkowo rzadkie, ale od czasu do czasu można się z nim spotkać. W odróżnieniu od innych rodzajów kodowania BaseX (16, 64, 36) w tym przypadku wykorzystywanych jest mniej niż dziesięć cyfr i w ogóle nie są używane litery. Używane są jedynie cyfry od 0 do 7. W językach programowania liczby ósemkowe są często reprezentowane za pomocą wiodącego zera — na przykład `017` to taka sama wartość jak `15` dziesiętnie lub `0F` szesnastkowo. Nie należy jednak zakładać, że wybrana liczba jest ósemkowa wyłącznie na podstawie wiodącego zera. Dane ósemkowe występują zbyt rzadko, aby na podstawie tej jednej wskazówki przyjmować takie założenie. Wiodące zera zwykle oznaczają stały rozmiar pola i niewiele poza tym. Kluczową cechą rozpoznawczą danych ósemkowych jest to, że składają się one z samych cyfr, z których żadna nie jest wartością większą od 7. Oczywiście ciąg `00000001` również pasuje do tego opisu, choć raczej nie są to dane ósemkowe. W rzeczywistości powyższy ciąg może być zapisany z użyciem dowolnego kodowania i nie ma to znaczenia. `1` zawsze oznacza `1`, niezależnie od kodowania!

Base36

Base36 to rzadko spotykana hybryda pomiędzy kodowaniem Base16 a Base64. Podobnie jak w przypadku Base16, cyfry rozpoczynają się od 0, a za cyfrą 9 są wykorzystywane w tej roli litery alfabetu. Ostatnią cyfrą w tym przypadku nie jest jednak `F`. W skład cyfr kodowania Base36 wchodzi wszystkie dwadzieścia sześć liter, aż do `Z`. Jednak w odróżnieniu od kodowania Base64 wielkość liter nie ma tu znaczenia oraz nie są wykorzystywane żadne znaki interpunkcyjne. A zatem jeśli zobaczymy mieszankę liter i cyfr, gdzie wszystkie litery będą wielkie bądź małe oraz gdzie będą występowały litery alfabetu spoza `F`, będzie to prawdopodobnie liczba zapisana z użyciem kodowania Base36.

Co powinniśmy wiedzieć o kodowaniu Base36?

Najważniejszą rzeczą, którą należy wiedzieć o kodowaniu Base36, podobnie jak w przypadku innych systemów liczenia, jest fakt, iż jest to liczba, pomimo że wygląda jak dane. Podczas wyszukiwania problemów związanych z przewidywalnymi i sekwencyjnymi identyfikatorami (omówimy je w recepturze 9.4) powinniśmy pamiętać, że następna wartość za `9X67DFR` to `9X67DFS`, natomiast o jeden niższa to `9X67DFQ`. Kiedyś spotkaliśmy się ze sklepem internetowym, w którym dzięki manipulowaniu parametrami zapisanymi z wykorzystaniem kodowania Base36 przekazywanymi w adresie URL udało się nam uzyskać dziewięćdziesięcioprocentowy rabat!

Dyskusja

Znalezienie narzędzia do kodowania Base16 i Base8 jest bardzo proste. Do tego celu można posłużyć się nawet prostym kalkulatorem w systemie Windows. Znalezienie narzędzia kodowania (dekodowania) dla standardu Base36 jest jednak nieco trudniejsze.

4.2. Korzystanie z danych Base64

Problem

Kodowanie Base64 wypełnia bardzo szczególną niszę: pozwala na kodowanie danych binarnych, które są niedrukowalne lub nie są bezpieczne dla kanału, w którym są przesyłane. Dane są kodowane do postaci stosunkowo nieczytelnej dla człowieka i bezpiecznej do transmisji za pomocą wyłącznie znaków alfanumerycznych i kilku znaków interpunkcyjnych. Często można spotkać złożone parametry zakodowane w Base64. W związku z tym bardzo potrzebna jest umiejętność ich dekodowania, modyfikowania i ponownego kodowania.

Rozwiązanie

Należy zainstalować OpenSSL w środowisku Cygwin (w systemie Windows) lub upewnić się, że mamy dostęp do polecenia `openssl` w przypadku korzystania z innego systemu operacyjnego. Pakiet OpenSSL występuje we wszystkich znanych dystrybucjach systemu Linux i Mac OS X.

Dekodowanie ciągu

```
% echo 'Q29uZ3JhdHVsYXRpb25zIQ==' | openssl base64 -d
```

Kodowanie całej zawartości pliku

```
% openssl base64 -e -in input.txt -out input.b64
```

Wykonanie powyższego polecenia spowoduje umieszczenie wyniku zakodowanego w Base64 w pliku o nazwie *input.b64*.

Kodowanie prostego ciągu znaków

```
% echo -n '&a=1&b=2&c=3' | openssl base64 -e
```

Dyskusja

Z kodowaniem Base64 można się spotkać bardzo często. Wykorzystuje się je w wielu nagłówkach HTTP (na przykład w nagłówku `Authorization:`). Także większość wartości przesyłanych w plikach cookie jest kodowana za pomocą Base64. Również wiele aplikacji koduje złożone parametry za pomocą Base64. Jeśli zobaczymy kodowane dane, zwłaszcza zawierające znaki równości, najpierw powinniśmy założyć, że są to dane Base64.

Zwróćmy uwagę na opcję `-n` w instrukcji `echo`. W taki sposób wyłącza się dodawanie znaku przejścia do nowego wiersza na końcu ciągu znaków przekazanego jako argument. Jeśli nie wyłączy się dodawania znaku przejścia do nowego wiersza, stanie się on częścią wyniku. W listingu 4.1 zamieszczono dwa różne polecenia wraz z odpowiadającymi im wynikami działania.

Listing 4.1. Wbudowane znaki przejścia do nowego wiersza w ciągach znaków zakodowanych z użyciem standardu Base64

```
% echo -n '&a=1&b=2&c=3' | openssl base64 -e    # Prawidłowo.
JmE9MSZiPTImYzOz

% echo '&a=1&b=2&c=3' | openssl base64 -e       # Nieprawidłowo.
JmE9MSZiPTImYzOzCg==
```

Niebezpieczeństwo występuje także wtedy, gdy wstawimy dane binarne do pliku, a następnie skorzystamy z opcji `-in` w celu zakodowania całego pliku. Prawie wszystkie edytory dodają znak przejścia do nowego wiersza na końcu ostatniego wiersza w pliku. Jeśli nie o to nam chodzi (ponieważ plik zawiera dane binarne), to powinniśmy zachować szczególną ostrożność podczas tworzenia danych wejściowych.

Dla wielu czytelników może być zaskakujące to, że do kodowania danych z wykorzystaniem Base64 używamy OpenSSL, skoro wyraźnie widać, że nie ma tu SSL ani innego szyfrowania. Polecenie `openssl` jest w pewnym sensie szwajcarskim nożem wojskowym. Za jego pomocą można wykonać wiele operacji, nie tylko kryptograficznych.

Rozpoznawanie kodowania Base64

W kodowaniu Base64 wykorzystuje się wszystkie znaki alfabetu, wielkie i małe litery oraz cyfry 0 – 9. W sumie daje to sześćdziesiąt dwa znaki. Ponadto wykorzystuje się znaki plusa (+) oraz ukośnika (/), co w sumie daje sześćdziesiąt cztery znaki. Znak równości również należy do zestawu dostępnych znaków, ale dodaje się go wyłącznie na końcu. Ciągi zakodowane w Base64 zawsze zawierają liczbę znaków podzielną przez 4. Jeśli dane wejściowe po zakodowaniu nie zawierają liczby bajtów podzielnej przez 4, dodaje się jeden lub kilka znaków równości (=), tak by uzyskać liczbę znaków będącą wielokrotnością 4. Tak więc w ciągu zakodowanym w Base64 będą występowały maksymalnie trzy znaki równości, choć może ich tam nie być wcale bądź może występować tylko jeden lub dwa znaki. Co więcej, jest to jedyne kodowanie, w którym wykorzystuje się kombinację wielkich i małych liter alfabetu.

Należy pamiętać o tym, że Base64 to kodowanie. Nie jest to szyfrowanie (ponieważ można je w prosty sposób odwrócić, bez konieczności wykorzystania specjalnych kluczy). Jeśli zetkniemy się z bardzo ważnymi danymi (na przykład poufnymi danymi, danymi mającymi wpływ na bezpieczeństwo, danymi do zarządzania programami) zakodowanymi w Base64, powinniśmy traktować je tak samo, jakby były zapisane zwykłym tekstem. Biorąc to pod uwagę, Czytelnik może założyć swój czarny hakerski kapelusz i zapytać siebie, co zyskuje, potrafiąc czytać zakodowane dane.

Zwróć również uwagę na to, że w danych zakodowanych w Base64 nie wykorzystuje się kompresji. Wręcz przeciwnie, zakodowane dane zawsze mają większą objętość od niezakodowanych. Może to stwarzać problemy, na przykład podczas projektowania bazy danych. Jeśli zmienimy w programie sposób przechowywania identyfikatorów użytkownika — z danych w postaci zwykłego tekstu (na przykład o maksymalnym rozmiarze ośmiu znaków) na dane zakodowane w Base64 — będziemy zmuszeni do zwiększenia rozmiaru pola do dwunastu znaków. Może to mieć istotny wpływ na projekt całego systemu — jest to zatem dobre miejsce do przeprowadzania testów zabezpieczeń.

Inne narzędzia

W tym przykładzie posłużyliśmy się OpenSSL, ponieważ jest to program szybki, niewielki i łatwo dostępny. Kodowanie i dekodowanie w standardzie Base64 można również z łatwością wykonać za pomocą programu CAL9000. Należy postępować zgodnie z instrukcjami zamieszczonymi w recepturze 4.5, ale wybrać *Base64* jako typ kodowania lub dekodowania. Także w przypadku korzystania z programu CAL9000 powinniśmy się zabezpieczyć przed przypadkowym wklejaniem znaków przejścia do nowego wiersza w polach tekstowych.

Można również skorzystać z modułu `MIME::Base64` dla języka Perl. Chociaż nie jest to moduł standardowy, z pewnością większość czytelników ma go w swoim systemie, ponieważ instaluje się on razem z modułem LibWWWPerl, który omówimy w rozdziale 8.

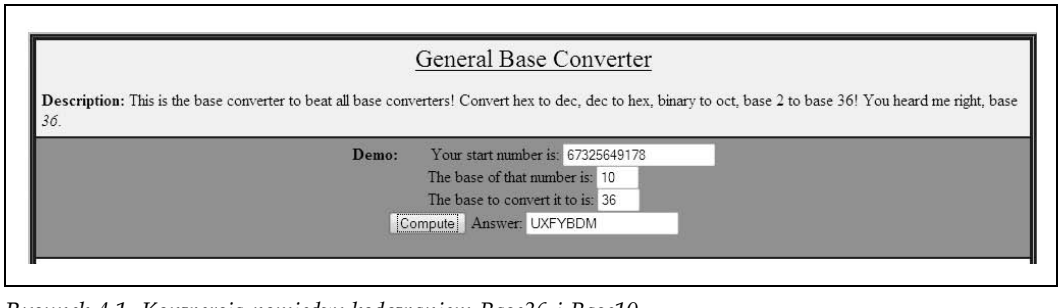
4.3. Konwersja liczb zakodowanych w Base36 na stronie WWW

Problem

Potrzebujemy zakodować lub zdekodować liczby Base36, a nie chcemy pisać w tym celu skryptu lub programu. Sposób zaprezentowany w tej recepturze jest prawdopodobnie najłatwiejszym sposobem okazjonalnej konwersji liczb zapisanych w różnych systemach kodowania.

Rozwiązanie

Brian Risk stworzył demonstracyjny serwis WWW pod adresem *http://www.geneffects.com/briarskin/programming/newJSMathFuncs.html*. Można w nim przeprowadzać dowolne konwersje z jednego systemu kodowania na inny. Aby przeprowadzić konwersję z kodowania Base10 na Base36 (lub odwrotnie), wystarczy wprowadzić wartości podstaw systemów kodowania w odpowiednich polach formularza. Przykład konwersji dużej liczby Base10 na Base36 pokazano na rysunku 4.1. Aby przeprowadzić konwersję z kodowania Base36 na Base10, wystarczy zamienić wartości `10` i `36` na stronie WWW.



Rysunek 4.1. Konwersja pomiędzy kodowaniem Base36 i Base10

Dyskusja

Fakt, że konwersja jest wykonywana w przeglądarce, nie oznacza, że w celu jej przeprowadzenia trzeba być podłączonym do internetu. Można zapisać kopię tej strony na lokalnym dysku twardym i załadować ją w przeglądarce w momencie, gdy zajdzie potrzeba wykonania konwersji (analogicznie jak w przypadku programu CAL9000 — zobacz: receptura 4.5).

4.4. Korzystanie z danych Base36 w Perlu

Problem

Mamy potrzebę kodowania lub dekodowania dużej ilości danych w standardzie Base36. Na przykład jest wiele liczb, które należy poddać konwersji, lub trzeba przeprowadzić programowe testowanie.

Rozwiązanie

Spośród narzędzi zaprezentowanych w niniejszej książce do tego zadania najbardziej nadaje się Perl. Zawiera bibliotekę `Math::Base36`, którą można zainstalować za pomocą repozytorium CPAN lub z wykorzystaniem standardowej metody instalacji modułów ActiveState (patrz: rozdział 2.). Sposób kodowania i dekodowania liczb w standardzie Base36 pokazano w listingu 4.2.

Listing 4.2. Skrypt Perl do konwersji liczb Base36

```
#!/usr/bin/perl
use Math::Base36 qw(:all);

my $base10num = 67325649178; # Po konwersji powinna przyjąć postać UXFYBDM
my $base36num = "9FFGK4H";   # Po konwersji powinna przyjąć postać 20524000481

my $newb36    = encode_base36( $base10num );
my $newb10    = decode_base36( $base36num );

print "b10 $base10num\t= b36 $newb36\n";
print "b36 $base36num\t= b10 $newb10\n";
```

Dyskusja

Więcej informacji na temat modułu `Math::Base36` można uzyskać za pomocą polecenia **`perldoc Math::Base36`**. Jedną z możliwości, jaką oferuje moduł, jest wypełnienie liczb dziesiętnych wiodącymi zerami z lewej strony.

4.5. Wykorzystanie danych kodowanych w URL

Problem

W danych kodowanych w URL wykorzystuje się znak `%` i cyfry szesnastkowe po to, by przesyłać w adresie URL dane, których nie można przesyłać tam bezpośrednio. Kilka przykładów

znaków tego typu to spacja, nawiasy trójkątne (< i >) oraz ukośnik (/). Jeśli w aplikacji internetowej występują dane kodowane w URL (na przykład w postaci parametrów, danych wejściowych lub kodu źródłowego), które chcemy zrozumieć bądź przetworzyć, musimy najpierw je zdekodować bądź zakodować.

Rozwiązanie

Najprościej operacje te można wykonać za pomocą programu CAL9000 grupy OWASP. Jest to seria stron WWW w HTML, które wykorzystują JavaScript do wykonywania podstawowych obliczeń. Za ich pomocą można interaktywnie kopiować i wklejać dane oraz kodować je i dekodować na żądanie.

Kodowanie

Należy wprowadzić zdekodowane dane w polu *Plain Text*, a następnie kliknąć opcję *Url (%XX)* znajdującą się z lewej strony w obszarze *Select Encoding Type*. Ekran aplikacji z wynikami tej operacji pokazano na rysunku 4.2.

Rysunek 4.2. Kodowanie URL za pomocą narzędzia CAL9000

Dekodowanie

Należy wprowadzić zakodowane dane w polu *Encoded Text*, a następnie kliknąć opcję *Url (%XX)* znajdującą się z lewej strony w obszarze *Select Decoding Type*. Ekran aplikacji z wynikami tej operacji pokazano na rysunku 4.3.

Rysunek 4.3. Dekodowanie danych zakodowanych w URL za pomocą narzędzia CAL9000

Dyskusja

Dane kodowane wewnątrz adresu URL powinny być znane wszystkim osobom, które kiedykolwiek oglądały kod źródłowy HTML lub dowolne dane przesyłane z przeglądarki WWW do serwera WWW. Ten sposób kodowania zapisano w dokumencie RFC 1738 (*ftp://ftp.isi.edu/in-notes/rfc1738.txt*). Standard ten nie wymaga kodowania niektórych znaków ASCII. Warto zwrócić uwagę na to, że chociaż nie jest to obowiązkowe, nic nie stoi na przeszkodzie, by kodować te znaki. Przykład pokazano w zakodowanych danych na rysunku 4.3. Nadmiarowe kodowanie to jeden ze sposobów, w jaki napastnicy maskują złośliwe dane wejściowe. Nieskomplikowane systemy „czarnych list" sprawdzające występowanie ciągu `<script>` lub nawet `%3cscript%3e` mogą nie zauważyć ciągu `%3c%73%63%72%69%70%74%3e`, który oznacza dokładnie to samo co dwa poprzednie.

Jedną z doskonałych własności programu CAL9000 jest fakt, iż w rzeczywistości nie jest to program. Jest to raczej kolekcja stron WWW zawierających osadzony kod JavaScript. Nawet jeśli w jakiejś firmie jest stosowana drakońska polityka zabraniająca instalowania czegokolwiek na stacjach roboczych, można przecież otworzyć strony WWW w przeglądarce i uruchomić odpowiednie funkcje. Strony WWW można bez trudu zapisać na dysku USB i załadować je bezpośrednio z niego, dzięki czemu nie ma potrzeby instalowania czegokolwiek.

4.6. Wykorzystywanie danych w formacie encji HTML

Problem

Specyfikacja HTML zapewnia sposób kodowania znaków o specjalnym znaczeniu, tak by nie były interpretowane jako HTML, JavaScript czy innego rodzaju polecenia. Aby można było generować przypadki testowe i przeprowadzać potencjalne ataki, trzeba umieć kodować i dekodować dane zgodnie z tym standardem.

Rozwiązanie

Kodowanie i dekodowanie tego typu najłatwiej przeprowadzić za pomocą narzędzia CAL9000. Nie będziemy tu zamieszczać szczegółowych instrukcji posługiwania się programem CAL9000, ponieważ jest to narzędzie, którego używa się w dość prosty sposób. Szczegółowe informacje na ten temat można znaleźć w recepturze 4.5.

W celu zakodowania specjalnych znaków należy wprowadzić specjalne znaki w polu *Plain Text* i wybrać swoje kodowanie. W polu *Trailing Characters* w programie CAL9000 należy wprowadzić średnik (;).

Dekodowanie znaków zakodowanych w postaci encji HTML wykonuje się tak samo, ale w odwróconej kolejności. Należy wpisać lub wkleić zakodowane dane w polu *Encoded Text*, a następnie kliknąć opcję *HTML Entity* znajdującą się z lewej strony, w obszarze *Select Decoding Type*.

Dyskusja

Kodowanie za pomocą encji HTML to obszar, w którym można popełnić wiele potencjalnych pomyłek. W naszej pracy spotykaliśmy się z wieloma przypadkami, w których w pewnych miejscach aplikacji stosowano kodowanie encji HTML (na przykład znak ampersand był kodowany jako `&amp;`), a w innych nie. Ważne jest nie tylko to, aby kodowanie było wykonywane prawidłowo. Okazuje się, że ze względu na występowanie wielu odmian kodowania encji HTML napisanie aplikacji internetowej, która właściwie obsługuje kodowanie, jest bardzo trudne.

Różne odmiany encji HTML

Istnieje co najmniej pięć lub sześć prawidłowych i stosunkowo dobrze znanych sposobów kodowania tego samego znaku za pomocą encji HTML. Kilka możliwości kodowania tego samego znaku — symbolu „mniejszy niż" (<) — zaprezentowano w tabeli 4.1.

Tabela 4.1. Różne odmiany kodowania encji

Odmiana kodowania	Zakodowany znak
Encje identyfikowane przez nazwę	&lt;
Wartości dziesiętne (ASCII lub ISO-8859-1)	&#60;
Wartości szesnastkowe (ASCII lub ISO-8859-1)	&#x3c;
Wartości szesnastkowe (długa liczba całkowita)	&#x003c;
Wartości szesnastkowe (liczby całkowite sześćdziesięcioczterobitowe)	&#x0000003c;

Istnieje nawet kilka dodatkowych metod kodowania specyficznych dla przeglądarki Internet Explorer. Z punktu widzenia możliwości testowania, jeśli mamy do przetestowania wartości graniczne lub specjalne, mamy do sprawdzenia co najmniej sześć do ośmiu permutacji: dwie lub trzy wersje kodowania w adresie URL oraz cztery lub pięć wersji kodowania za pomocą encji HTML.

Diabeł tkwi w szczegółach

Obsługa kodowania jest bardzo trudna dla programisty aplikacji z wielu powodów. Na przykład występuje wiele różnych miejsc, w których trzeba wykonywać kodowanie i dekodowanie, oraz istnieje wiele niezwiązanych ze sobą komponentów, które wykonują funkcje kodowania i dekodowania. Weźmy pod uwagę przypadek najbardziej popularny — proste żądanie `GET`. W pierwszej kolejności kodowaniem danych zajmuje się przeglądarka WWW. Przeglądarki różnią się jednak pomiędzy sobą kilkoma szczegółami. Następnie serwer WWW (na przykład IIS lub Apache) wykonuje kodowanie na danych wchodzących w stosunku do tych znaków, które nie zostały zakodowane przez przeglądarkę WWW. W dalszej kolejności na każdej platformie, na której uruchamiany jest kod, podejmowane są próby interpretacji, kodowania lub dekodowania niektórych strumieni danych. Na przykład w środowiskach webowych .Net i Java kodowanie URL i encje HTML są w większości obsługiwane niejawnie. Na koniec sama aplikacja może kodować bądź dekodować dane zapisane w bazie danych, pliku lub pamięci trwałej innego rodzaju. Próba zapewnienia tego, aby dane pozostawały zakodowane w prawidłowej formie w całej sekwencji wywołań (od przeglądarki do aplikacji), jest, mówiąc najbardziej ogólnie, bardzo trudna. Równie trudna jest analiza przyczyn wystąpienia problemów.

4.7. Wyliczanie skrótów

Problem

Kiedy aplikacja korzysta ze skrótów (ang. *hash*), sum kontrolnych lub innych sposobów kontroli integralności danych, trzeba umieć je rozpoznawać i ewentualnie je wyliczać w odniesieniu do danych testowych. Osobom, dla których pojęcie skrótów nie jest znane, polecam zapoznanie się z ramką „Czym są skróty?" w dalszej części tego rozdziału.

Rozwiązanie

Tak jak w przypadku innych zadań związanych z kodowaniem, do wyboru mamy co najmniej trzy dobre możliwości: OpenSSL, CAL9000 i Perl.

MD5

% echo -n "my data" | openssl md5

```
c:\> type myfile.txt | openssl md5
```

SHA-1

#/usr/bin/perl

```
use Digest::SHA1  qw(sha1);
$data   = "my data";
$digest = sha1($data);
print "$digest\n";
```

Czym są skróty

Skrót to jednokierunkowe przekształcenie matematyczne. Niezależnie od ilości danych wejściowych wynik ma zawsze taki sam rozmiar. Skróty silne pod względem kryptograficznym — takich używa się w większości istotnych funkcji zabezpieczeń — charakteryzują się kilkoma ważnymi właściwościami:

- odpornością na odgadnięcie przeciwobrazu (ang. *preimage resistance*): dla kogoś, kto wejdzie w posiadanie skrótu, znalezienie dokumentu bądź danych wejściowych, które generują ten skrót, powinno być trudne;
- odpornością na kolizje: dysponując określonym dokumentem lub danymi wejściowymi powinno być trudne znalezienie innego dokumentu lub danych wejściowych, które generują taki sam skrót.

W obydwu tych właściwościach mówimy, że wykonanie określonej operacji powinno być „trudne". Oznacza to, że pomimo iż jest to teoretycznie możliwe, powinno być to na tyle czasochłonne i na tyle mało prawdopodobne, aby napastnik zrezygnował z danej właściwości skrótu do przeprowadzenia praktycznego ataku.

Dyskusja

Skróty MD5 zaprezentowano z wykorzystaniem pakietu OpenSSL w systemie Unix lub Windows. W OpenSSL jest również funkcja `sha1` obsługująca skróty SHA-1. Zwróćmy uwagę na konieczność użycia opcji `-n` w uniksowej instrukcji `echo`, aby zabezpieczyć się przed dodawaniem znaku przejścia do nowego wiersza na końcu danych. Chociaż w systemie Windows również występuje polecenie `echo`, nie można wykorzystywać go tak samo jak w środowisku Unix, ponieważ nie pozwala ono na pomijanie zestawu znaków CR/LF na końcu komunikatu przekazywanego do niego w formie argumentu.

Przypadek zastosowania skrótów SHA-1 zaprezentowano na przykładzie skryptu Perla korzystającego z modułu `Digest::SHA1`. W Perlu jest również moduł `Digest::MD5`, który działa tak samo dla skrótów MD5.

Zwróć uwagę na to, że nie ma możliwości dekodowania skrótów. Skróty są przekształceniami matematycznymi, które działają tylko w jedną stronę. Niezależnie od ilości danych wejściowych wynik ma zawsze taki sam rozmiar.

Skróty MD5

Skróty MD5 generują dokładnie 128 bitów (16 bajtów) danych. Skróty MD5 można zaprezentować na kilka różnych sposobów:

32 cyfry szesnastkowe
`df02589a2e826924a5c0b94ae4335329`

24 znaki Base64
`PlnPFeQx5Jj+uwRfh//RSw==`. W takiej postaci skróty MD5 występują w przypadku, gdy skrót MD5 w postaci binarnej (128 bitów binarnych) zostanie zakodowanych w standardzie Base64.

Skróty SHA-1

Skróty SHA-1 zawsze generują dokładnie 160 bitów (20 bajtów) danych. Podobnie jak w przypadku skrótów MD5, można je zaprezentować na kilka różnych sposobów:

40 cyfr szesnastkowych
`bc93f9c45642995b5566e64742de38563b365a1e`

28 znaków Base64
`9EkBWUsXoiwtICqaZp2+VbZaZdI=`

> **Skróty a bezpieczeństwo**
>
> Częstym błędem w zabezpieczeniach aplikacji jest założenie, że zapisywanie lub przesyłanie haseł w postaci skrótów jest bezpieczne. Skróty są również często wykorzystywane w odniesieniu do kart kredytowych, numerów NIP oraz innych prywatnych danych. Problem z takim podejściem z punktu widzenia bezpieczeństwa polega na tym, że skrótów można użyć w taki sam sposób jak haseł, które one reprezentują. Jeśli do uwierzytelniania aplikacji wykorzystuje się identyfikator użytkownika oraz skrót SHA-1 hasła, aplikacja w dalszym ciągu może być narażona na niebezpieczeństwo. Do uwierzytelnienia napastnikowi może wystarczyć przechwycenie i użycie skrótu hasła (choć same hasło pozostanie dla niego tajemnicą). Należy podchodzić sceptycznie do skrótów haseł bądź innych wrażliwych informacji. Często napastnik nie musi znać informacji w postaci zwykłego tekstu — wystarczy, że przechwyci skrót hasła i odpowiednio go użyje.

4.8. Rozpoznawanie formatów czasowych

Problem

Czas może być reprezentowany na wiele różnych sposobów. Umiejętność rozpoznawania reprezentacji czasu pozwala na budowanie lepszych przypadków testowych. W pisaniu ukierunkowanych przypadków testowych pomaga nie tylko umiejętność rozpoznania, że określone dane oznaczają czas, ale także znajomość podstawowych założeń, jakie przyjął programista podczas pisania kodu.

Rozwiązanie

W najbardziej oczywistych formatach czasowych jest kodowany rok, miesiąc i dzień. Dane te występują w popularnych układach, przy czym rok jest reprezentowany za pomocą dwóch lub czterech cyfr. W niektórych formatach czasu występują godziny, minuty i sekundy, a czasami

dziesiąte części sekund i milisekundy. Kilka reprezentacji daty 1 czerwca 2008, 17:32:11 i 844 milisekundy pokazano w tabeli 4.2. W niektórych formatach określone części daty bądź godziny nie są reprezentowane. Te fragmenty są pomijane.

Tabela 4.2. Różne reprezentacje czasu

Kodowanie	Przykładowy wynik
YYYYMMDDhhmmss.sss	20080601173211.844
YYMMDDhhmm	0806011732
Czas Unix (liczba sekund od 1 stycznia 1970)	1212355931
POSIX wg standardu „C"	Nie 1 Cze 17:32:11 2008

Dyskusja

Na pozór można by sądzić, że rozpoznawanie czasu jest dość oczywistą umiejętnością i nie jest ważne dla kogoś, kto testuje aplikacje internetowe. Jesteśmy zdania, że jest to bardzo ważne. Spotykaliśmy się z wieloma aplikacjami, gdzie projektanci uważali czas za informację, której nie da się odgadnąć. Używano go w identyfikatorach sesji, tymczasowych nazwach plików, tymczasowych hasłach i numerach kont. Osoba przeprowadzająca symulowane ataki powinna wiedzieć, że czasu nie wolno uznać za nieprzewidywalny. Planując „interesujące" przypadki testowe dla określonego pola wejściowego, można znacznie zawęzić zbiór dopuszczalnych wartości testowych, jeśli się wie, że informacje te dotyczą czasu z niedawnej przeszłości lub z najbliższej przyszłości.

Milisekundy a losowość

Nie dajmy się nikomu przekonać, że wartości wyrażone w milisekundach są nieprzewidywalne. Intuicyjnie można oczekiwać, że nikt nie będzie w stanie przewidzieć, kiedy użytkownik prześle żądanie do serwera WWW. W związku z tym, jeżeli program czyta zegar i wyodrębnia z tej wartości tylko milisekundy, każda z tysiąca możliwości (0 – 999) powinna być jednakowo prawdopodobna. Intuicja podpowiada nam „tak", ale prawdziwa odpowiedź brzmi „nie". Okazuje się, że niektóre wartości są znacznie bardziej prawdopodobne od innych. Z różnych względów (na przykład dokładność odmierzania odcinków czasu przez jądro systemu operacyjnego — zarówno Unix, jak i Windows — dokładność zegara, przerwania i wiele innych) zegar jest bardzo złym generatorem liczb losowych. Znacznie dokładniejszy opis tego zjawiska zamieszczono w rozdziale 10. książki autorstwa Johna Viega i Gary'ego McGrawa *Building Secure Software* (Addison-Wesley).

Tester nie powinien ufać żadnemu systemowi oprogramowania, który do generowania losowych wartości wykorzystuje czas. Jeśli odkryjemy takie elementy w testowanych programach, powinniśmy natychmiast rozważać takie kwestie jak: „A co się stanie, jeśli komuś uda się odgadnąć tę wartość" lub „Jak zachowa się aplikacja, jeśli dwie pozornie losowe wartości okażą się takie same?".

4.9. Programowe kodowanie wartości oznaczających czas

Problem

Ustaliliśmy, że w naszej aplikacji jest wykorzystywany czas w interesujący sposób. Chcemy teraz wygenerować specyficzne wartości w specyficznych formatach.

Rozwiązanie

Do wykonania tego zadania idealnie nadaje się Perl. Do wykonywania operacji na wartościach czasu w formacie systemu Unix będziemy potrzebowali modułu `Time::Local`. Będzie nam również potrzebny moduł `POSIX`, który udostępnia funkcję `strftime`. Oba są modułami standardowymi. W listingu 4.3 zaprezentowano cztery różne formaty czasu i sposoby manipulowania nimi.

Listing 4.3. Kodowanie różnych wartości czasowych w Perlu

```
#!/usr/bin/perl
use Time::Local;
use POSIX qw(strftime);
# 1 czerwca 2008, 17:32:11 i 844 milisekundy .
$year  = 2008;
$month = 5;      # Miesiące są numerowane, począwszy od 0!
$day   = 1;
$hour  = 17;     # w celu zapewnienia lepszej czytelności skorzystamy z 24-godzinnego
zegara
$min   = 32;
$sec   = 11;
$msec  = 844;

# Czas w formacie UNIX (liczba sekund od 1 stycznia 1970 roku)     1212355931
$unixtime = timelocal( $sec, $min, $hour, $day, $month, $year );
print "UNIX\t\t\t$unixtime\n";

#  Wypełniamy danymi kilka wartości (wday, yday, isdst), które będą potrzebne do wykonania funkcji strftime.
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime($unixtime);

#  YYYYMMDDhhmmss.sss   200806011732 11.844
# Wykorzystujemy funkcję strftime(), ponieważ uwzględnia ona numerowanie miesięcy od zera, które jest typowe dla Perla.
$timestring = strftime( "%Y%m%d%H%M%S",
    $sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst );
$timestring .= ".$msec";
print "YYYYMMDDhhmmss.sss\t$timestring\n";

# YYMMDDhhmm  0806011732
$timestring = strftime( "%y%m%d%H%M",
$sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst );
print "YYMMDDhhmm\t\t$timestring\n";

# POSIX według standardu języka "C" Nie Cze  1 17:32:11 2008
$gmtime = localtime($unixtime);
print "POSIX\t\t\t$gmtime\n";
```

Dyskusja

Aby dowiedzieć się więcej na temat możliwych sposobów formatowania czasu, można skorzystać z poleceń `perldoc Time::Local` lub `man strftime`.

Osobliwości obsługi czasu w Perlu

Chociaż Perl jest bardzo elastyczny i z całą pewnością jest dobrym narzędziem do wykonywania tego zadania, charakteryzuje się pewnymi osobliwościami. Podczas wykonywania operacji na wartościach czasowych podobnych do tych, które pokazaliśmy w powyższym przykładzie, należy zwrócić szczególną uwagę na wartości miesięcy. Z pewnych trudnych do wyjaśnienia powodów liczenie miesięcy rozpoczyna się od 0. Zgodnie z tym styczniowi odpowiada liczba 0, natomiast lutemu 1. Właściwość ta nie dotyczy dni. Pierwszy dzień miesiąca ma numer 1. Co więcej, należy zwrócić uwagę na sposób kodowania roku. Numer roku odpowiada liczbie lat, które upłynęły od roku 1900. Tak więc dla roku 1999 wartość ta wynosi 99, natomiast dla roku 2008 jest to liczba 108. Aby uzyskać prawidłowy numer roku, do tej wartości należy dodać 1900. Pomimo całego szumu wokół roku 2000 w dalszym ciągu można spotkać serwisy WWW, które pokazują daty typu 28-06-108.

4.10. Dekodowanie wartości ViewState języka ASP.NET

Problem

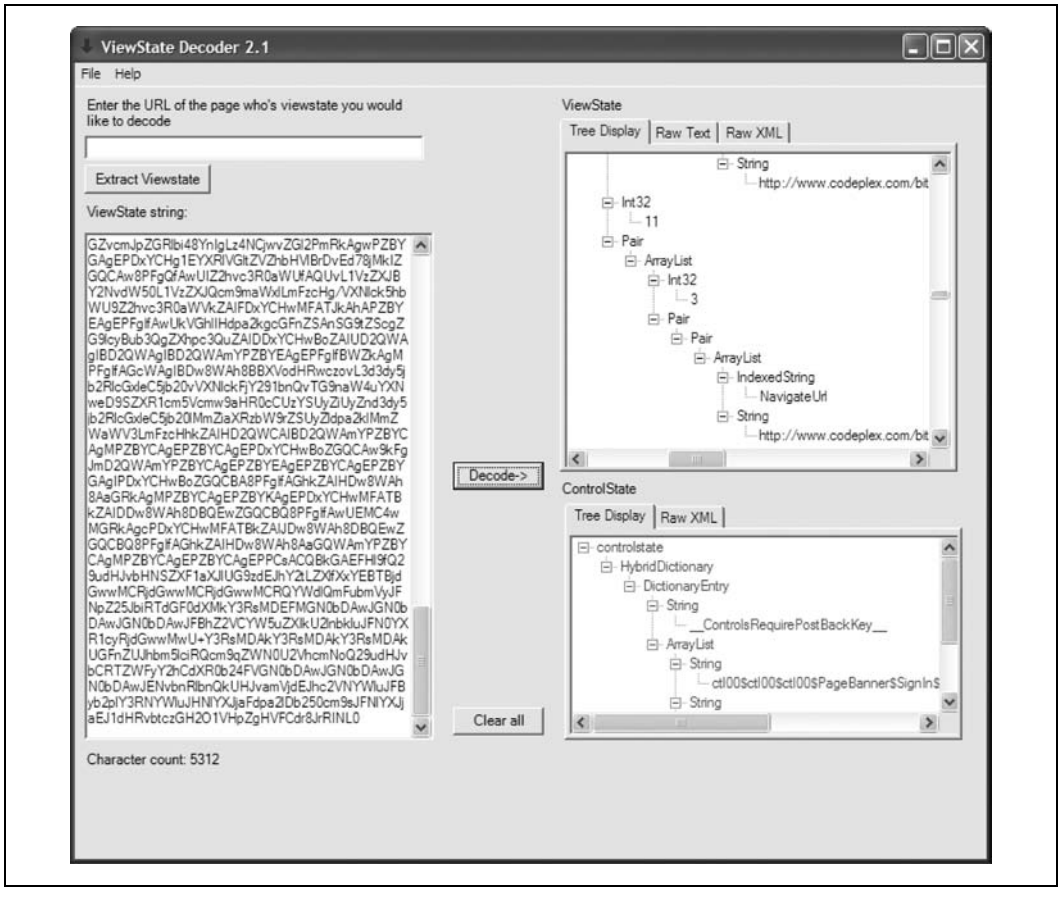
Język ASP.NET dostarcza mechanizmu, dzięki któremu stan może być zapisywany po stronie klienta zamiast po stronie serwera. Przeglądarka WWW może przesyłać z każdym żądaniem jako pola formularzy nawet stosunkowo rozbudowane obiekty opisu stanu (po kilka kilobajtów). Mechanizm ten nosi nazwę ViewState. Obiekt opisu stanu jest przechowywany jako pole wejściowe formularza `__VIEWSTATE`. Jeśli aplikacja korzysta z mechanizmu ViewState, to należy przeanalizować sposób, w jaki informacje przekazane tą drogą są wykorzystywane przez logikę reguł biznesu, i opracować testy obejmujące wykorzystanie zmodyfikowanych danych ViewState. Aby można było opracowywać testy wokół zmodyfikowanych danych ViewState, trzeba zrozumieć sposób posługiwania się danymi ViewState w aplikacji.

Rozwiązanie

Należy pobrać aplikację ViewState Decoder z witryny Fritz Onion (*http://www.pluralsight.com/tools.aspx*). Najprostszy sposób jej użycia polega na skopiowaniu adresu URL aplikacji (lub określonej strony) do adresu URL. Na rysunku 4.4 pokazano zrzut z ekranu pochodzący z wersji 2.1 aplikacji ViewState Decoder oraz niewielki fragment wyniku jej działania.

Dyskusja

Czasami programowi nie udaje się pobrać informacji ViewState ze strony WWW. W rzeczywistości nie jest to duży problem. Wystarczy przejrzeć źródło strony WWW (patrz: receptura 3.2) i poszukać ciągu `<input type= "hidden" name="__VIEWSTATE"…>`. Należy skopiować wartość tego pola wejściowego i wkleić do dekodera.



Rysunek 4.4. Dekodowanie danych przesyłanych za pomocą mechanizmu ViewState języka ASP.NET

Gdyby w przykładzie pokazanym na rysunku 4.4 była nasza aplikacja, można by na tej podstawie znaleźć kilka potencjalnych ścieżek testowania. W danych ViewState są adresy URL. Czy mogą one zawierać kod JavaScript lub kierować użytkownika do innego, złośliwego serwisu WWW? A co z różnymi liczbami całkowitymi?

Jeśli aplikacja wykorzystuje ASP.NET i mechanizm ViewState, należy odpowiedzieć sobie na kilka istotnych pytań:

- Czy jakiekolwiek dane z pola ViewState są wstawiane do adresu URL lub kodu HTML strony w czasie, gdy serwer ją przetwarza?

  Zwróćmy uwagę na adresy URL widoczne na rysunku 4.4. Co by się stało, gdyby łącza nawigacyjne do strony w tej aplikacji pochodziły z danych ViewState? Czy haker zdołałby nakłonić kogoś do wizyty w złośliwym serwisie WWW poprzez wysłanie mu „skażonych" informacji ViewState?

- Czy pole ViewState jest zabezpieczone przed możliwością modyfikowania?

  ASP.NET dostarcza kilku sposobów zabezpieczania pola ViewState. Jeden z nich to zastosowanie prostego skrótu. Dzięki niemu serwer może wykryć sytuację wyjątkową w przypadku nieoczekiwanej modyfikacji pola ViewState. Drugi to zastosowanie mechanizmu

szyfrującego, dzięki czemu pole ViewState stanie się nieczytelne z poziomu klienta oraz potencjalnego napastnika.

- Czy jakakolwiek część logiki programu ślepo polega na wartości odczytanej z pola ViewState?

  Wyobraźmy sobie aplikację, w której dane na temat typu użytkownika (zwykły bądź administrator) są zapisane w polu ViewState. Napastnik musiałby tylko zmodyfikować te dane, aby zmienić swoje prawa w aplikacji.

Podczas tworzenia testów bazujących na uszkodzonych danych ViewState do wstawiania nowych wartości należy wykorzystać takie narzędzia jak TamperData (patrz: receptura 3.6) lub WebScarab (patrz: receptura 3.4).

4.11. Dekodowanie danych zakodowanych wielokrotnie

Problem

Czasami dane są zakodowane wiele razy. Bywa, że jest to celowe, innym razem jest to efekt uboczny przekazywania danych przez oprogramowanie pośrednie. Na przykład w ciągach znaków zakodowanych w Base64 (patrz: receptura 4.2) często można spotkać znaki niealfanumeryczne (=, /, +), które są zakodowane według reguł obowiązujących dla adresów URL (patrz: receptura 4.5). Na przykład ciąg `V+P//z==` może wyświetlać się jako `V%2bP%2f%2f%3d%3d`. Należy o tym pamiętać i po zakończeniu jednego etapu skutecznego kodowania traktować wynik jako dane, które potencjalnie są zakodowane innym sposobem.

Rozwiązanie

Czasami pojedynczy parametr jest w rzeczywistości specjalną strukturą, w której jest zapisanych wiele parametrów. Na przykład jeśli zobaczymy ciąg `AUTH=dGVzdHVzZXI6dGVzdHB3MTIz`, może się nam wydawać, że `AUTH` jest jednym parametrem. Kiedy przekonamy się, że po zdekodowaniu wartość przyjmuje postać `testuser:testpw123`, zdamy sobie sprawę z tego, że w rzeczywistości jest to parametr złożony zawierający identyfikator użytkownika i hasło oddzielone od siebie dwukropkiem. W związku z tym w naszych testach musimy osobno przetwarzać składowe tej wartości. Reguły przetwarzania identyfikatorów użytkownika i haseł w aplikacjach internetowych niemal na pewno są różne.

Dyskusja

Zazwyczaj nie zamieszczamy ćwiczeń jako dodatków do receptur, ale w tym przypadku warto to zrobić. Rozpoznawanie kodowania danych jest dość ważną umiejętnością. Wykonanie kilku ćwiczeń może pomóc w ugruntowaniu zaprezentowanych informacji. Należy pamiętać, że niektóre z nich mogą być zakodowane więcej niż raz. Spróbujmy sprawdzić, czy potrafimy rozpoznać rodzaj kodowania w poniższych przypadkach (odpowiedzi zamieszczono w przypisach):

1. `xIThJBeIucYRX4fqS+wxtR8KeKk=`[1]
2. `TW9uIEFwciAgMiAyMjoyNzoyMSBFRFQgMjAwNwo=`[2]
3. `4BJB39XF`[3]
4. `F8A8OEE2F6484CF68B7B72795DD31575`[4]
5. `O723O345O556O231`[5]
6. `713ef19e569ded13f2c7dd379657fe5fbd44527f`[6]

[1] Skrót MD5 zakodowany w Base64.

[2] Skrót SHA-1 zakodowany w Base64.

[3] Base36.

[4] Skrót MD5 zapisany szesnastkowo.

[5] Liczba ósemkowa.

[6] Skrót SHA-1 zapisany szesnastkowo.