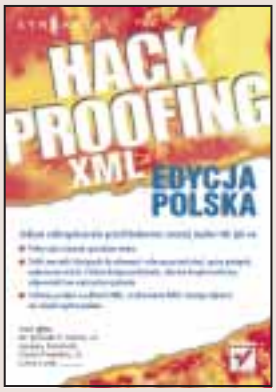
Hack Proofing XML. Edycja polska



Autor: praca zbiorowa
Tłumaczenie: Adam Jarczyk
ISBN: 83-7361-004-9
Tytuł oryginału: Hack Proofing XML
Format: B5, stron: 324
Przykłady na ftp: 15 kB

XML szybko staje się uniwersalnym protokołem wymiany informacji pomiędzy systemami używającymi HTTP. HTML zapewne zachowa swoją pozycję języka opisującego wygląd dokumentów w sieci WWW, jednak tam, gdzie w grę wchodzą dane, XML jest dużo lepszym rozwiązaniem. Walidacja, czyli sprawdzenie poprawności dokumentu XML, to pierwsza zapora przed atakami hakerskimi. Te same właściwości, które czynią XML silnym i uniwersalnym narzędziem sprawiają, że jest on podatny na działania hakerów. Wiele zapór sieciowych nie filtruje dokumentów XML – to kolejna przyczyna, dla której niepoprawne strukturalnie dokumenty mogą stanowić poważne zagrożenie dla systemów. „Hack Proofing XML. Edycja polska" objaśni Ci wszystkie niuanse bezpieczeństwa związane z technologiami XML i .NET.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Typ dokumentu — kontrola poprawności

Wprowadzenie

Definicja typu dokumentu (*DTD — document type definition*) i *schemat* (*Schema*) grają zasadniczą rolę w zapewnianiu poprawności dokumentu XML. Te dwa mechanizmy są ze sobą skojarzone na wiele sposobów, lecz każdy z nich spełnia odpowiednią funkcję w weryfikacji, czy dokument XML będzie zachowywał się zgodnie z oczekiwaniami. Właściwe wykorzystanie DTD i schematów pomaga programiście koncentrować się na projekcie struktury danych, zamiast martwić się o błędy pisowni i formy, spowalniając proces twórczy.

W niniejszym rozdziale najpierw zajmiemy się mechanizmami działania DTD i schematów dostępnych dla XML-a. Zobaczymy, czym różnią się DTD i schemat oraz jak mogą razem służyć do zapewnienia poprawności dokumentu. Następnie opiszemy ogólne zasady ataku tekstem jawnym oraz zakończymy rozdział kilkoma poradami, na co należy zwracać uwagę przy kontroli poprawności XML-a.

Kontrola poprawności (*validation*) dokumentu XML i komunikatów wysyłanych do niego jest pierwszą czynnością podczas zabezpieczania XML-a przed włamaniami. Właściwości, które czynią z XML-a potężny język służący do definiowania danych w dowolnych systemach powodują zarazem, iż jest on podatny na ataki. Co więcej, ponieważ wiele zapór firewall przepuszcza dane XML bez filtrowania, źle zbudowany i nie sprawdzony pod względem poprawności dokument XML może stanowić poważną lukę w zabezpieczeniach na poziomie systemu.

Definicje typu dokumentu i poprawnie zbudowane dokumenty XML

DTD są strukturalnymi narzędziami kontroli poprawności dokumentów XML. Zewnętrzne DTD mogą opisywać właściwości atrybutów, elementów i jednostek stosowanych w dokumencie XML. Do opisywanych właściwości należą zawartość, ilość (liczba) i struktura każdej pozycji. DTD mogą być częścią dokumentu lub zewnętrznymi obiektami względem używających je dokumentów. Definicjami DTD mogą być specjalnie stworzone opisy struktur danych, składniki specyfikacji stosowanych przez partnerów w biznesie lub też standardowe dokumenty używane przez autorów dokumentów XML na całym świecie.

Zanim będziemy mogli użyć DTD do sprawdzenia, czy dany dokument jest poprawnie zbudowany, musimy zdeklarować tę definicję. Deklaracja DTD dla prostej pozycji w katalogu może wyglądać następująco:

```
<?xml version="1.0"?><!DOCTYPE catalog [
     <!ELEMENT Katalog (Produkt*)>
     <!ELEMENT Produkt (NrProduktu*, NazwaProduktu*, CenaSklepowa*)>
     <!ELEMENT NrProduktu (#PCDATA)>
     <!ELEMENT NazwaProduktu (#PCDATA)>
     <!ELEMENT CenaSklepowa (#PCDATA)>

     <!ENTITY comment_outofstock "Ten towar jest już niedostepny w magazynie.">
]>
```

Powyższy fragment kodu DTD oznacza, iż katalog może zawierać dowolną liczbę wpisów produktów, aczkolwiek nie musi zawierać ani jednego. Każdy produkt może (lecz nie musi) posiadać elementy `NrProduktu`, `NazwaProduktu` i `CenaSklepowa`, które są danymi typu znakowego. Ponadto, zdefiniowaliśmy stałą łańcuchową, która zawsze zawiera komunikat o wyczerpaniu zapasów produktu. Proszę zwrócić uwagę, że DTD może ograniczyć typ danych zawartych w elemencie do danych znakowych, lecz nie wymusza określonego układu cyfr, liter i znaków sterujących, o ile nie definiuje translacji łańcucha znaków dla nazwy jednostki. Jak zobaczymy w dalszej części rozdziału, w tych kontekstach schematy pozwalają na o wiele dokładniejszą kontrolę niż DTD.

Poprzedni przykład mógł zdefiniować te same informacje w sposób nieco odmienny, definiując atrybuty elementu `Produkt`. DTD stworzony w ten sposób wyglądałby następująco:

```
<?xml version="1.0"?>

<!DOCTYPE catalog [
     <!ELEMENT Katalog (Produkt*)>
     <!ELEMENT Produkt EMPTY>

     <!ATTLIST Produkt NrProduktu CDATA #REQUIRED>
                       NazwaProduktu CDATA #REQUIRED>
                       CenaSklepowa CDATA #REQUIRED>

     <!ENTITY comment_outofstock "Ten towar jest już niedostepny w magazynie.">
]>
```

Powyższa definicja DTD mówi, iż element `Katalog` posiada jeden podelement — `Produkt`. Liczba produktów może być dowolna (również 0). `Produkt` nie posiada podelementów, jedynie trzy atrybuty:

- `NrProduktu`
- `NazwaProduktu`
- `CenaSklepowa`

Te trzy atrybuty muszą posiadać wartości, jeśli element `Produkt` istnieje. Definiowane informacje w obu przypadkach są takie same, lecz istnieją subtelne różnice w sposobach organizacji danych i kontroli danych potomnych przez element nadrzędny.

DTD nie są zapisywane zgodnie ze składnią dokumentu XML.

Spoglądając na proste DTD z przykładów, zauważymy, iż struktura języka różni się od normalnej składni XML-a. Oznacza to, że poprawność dokumentu DTD nie może być sprawdzana przez analizator składni kontrolujący poprawność XML-a. Jednym z powodów, dla których opracowano schematy, była chęć pozbycia się w XML-u potrzeby stosowania dwóch odrębnych gramatyk: jednej dla dokumentu XML, a drugiej dla narzędzia nadającego strukturę i sprawdzającego poprawność.

DTD mogą być albo wewnętrzne (zawarte w samym dokumencie XML) lub zewnętrzne (w serwerze, do którego dokument ma dostęp). Zewnętrzne DTD są powszechnie spotykane i często używane do wymuszenia określonej struktury danych lub jednolitej stylistyki dokumentów tworzonych przez różne wydziały lub jednostki partnerskie. Odwołanie do zewnętrznej definicji DTD wymaga użycia zewnętrznej deklaracji o następującej postaci:

```
<!DOCTYPE catalog SYSTEM "http://tempuri.org/Catalog1.dtd">
```

Kilka podstawowych deklaracji i atrybutów pokrywa ogromną większość instrukcji spotykanych w DTD, zarówno wewnętrznych, jak i zewnętrznych. Tabela 4.1 przedstawia najważniejsze typy atrybutów i ich zastosowania. Tabela 4.2 zawiera najprzydatniejsze deklaracje elementów i ich właściwości. Tabela 4.3 wymienia najczęściej stosowane atrybuty DTD i ich definicje.

Tabela 4.1. *Typy i zastosowanie atrybutów DTD*

Typ atrybutu	Zastosowanie atrybutu	Właściwości atrybutu
`CDATA`	`<!ATTLIST nazwa CDATA>`	Dane znakowe. Może zawierać znaki (<), znaki przedstawione jako nazwy (`<`) lub numery (`&60;`)
`ENTITY`	`<!ENTITY foto1 SYSTEM "c:\foto1.jpg">`	Odwołanie do obiektu, który nie będzie analizowany składniowo. Poprzez deklaracje `Entity` są często tworzone odwołania do plików graficznych lub multimedialnych.

Tabela 4.1. *Typy i zastosowanie atrybutów DTD — ciąg dalszy*

Typ atrybutu	Zastosowanie atrybutu	Właściwości atrybutu
Enumeration	<!ATTLIST cegla (licowka | fakturowana) #REQUIRED>	Lista atrybutów. Atrybuty rozdzielone przez znak | muszą być pobierane pojedynczo.
ID	<!ATTLIST Mlotek SKU ID #REQUIRED>	Wartość atrybutu musi być legalną nazwą XML. Musi też być unikatowa w obrębie dokumentu. Przypomina to stosowanie atrybutu klucza w bazach danych.
IDREF	<!ATTLIST Narzedzia MlotekSKU IDREF #REQUIRED>	Wartość atrybutu jest identyfikatorem innego elementu i musi się ściśle zgadzać (proszę pamiętać o rozróżnianiu wielkości liter). Ten atrybut służy do wywoływania identyfikatorów zdeklarowanych w atrybucie ID.
NMTOKEN	<!ATTLIST klucz NMTOKEN #REQUIRED>	Wartość atrybutu musi być legalną nazwą XML. W tym przypadku nazwa nie ma specjalnej funkcji ani możliwości, lecz funkcjonuje przede wszystkim jako etykieta. Może być to przydatne przy przekazywaniu poprzez dokument XML informacji do języka programowania, np. Javy.
NOTATION	<!ATTLIST typograficzny NOTATION #REQUIRED>	Kolejna metoda wskazania na plik lub inny zasób, np. zawartość multimedialną, nie podlegająca analizie składni.

Tabela 4.2. *Deklaracje i właściwości elementów DTD*

Deklaracje elementów	Właściwości deklaracji
#PCDATA	Analizowane składniowo dane znakowe. Podobny do typu atrybutu CDATA. Musi zawierać jedynie znaki.
ANY	Oznacza, że element może zawierać dane dowolnego typu.
Choices	Element może zawierać dowolny z listy elementów potomnych. Elementy potomne rozdzielone przecinkami mogą być wszystkie obecne. Gdy elementy rozdzielone są znakiem (|), wówczas może być obecny jeden, lecz nie wszystkie.

Tabela 4.3. *Atrybuty DTD i ich definicje*

Atrybut domyślny	Definicja atrybutu
#Fixed	Atrybut będzie miał wartość zdefiniowaną w deklaracji. Wartość ta nie może zostać zmieniona i pozostaje stała przez cały czas działania aplikacji.
#Implied	Atrybut opcjonalny. Zdefiniowany element może pozostać pusty bez skutków ubocznych.
Literal	Atrybut posiada wartość początkową (domyślną) podaną w deklaracji. Wartość ta może zostać zmieniona przez dane wejściowe lub działanie aplikacji.
#Required	Do atrybutu musi być przypisana wartość.

Schemat i poprawne dokumenty XML

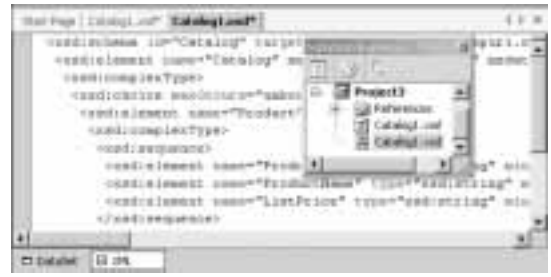
Każdy dokument XML, aby prawidłowo funkcjonować, musi być *poprawnie zbudowany* (*well-formed*) i *poprawny* (*Valid*). Są to dwie niezależne nazwy dwóch całkiem odrębnych właściwości dokumentu. Poprawnie zbudowany dokument XML może nie być poprawnym dokumentem XML, lecz dokument XML nie zbudowany poprawnie nie może być poprawny. Poprawnie zbudowany dokument XML spełnia określone wymagania dotyczące znaczników elementu głównego, znaczników początkowych i końcowych, elementów i atrybutów oraz dopuszczalnych znaków. Poprawna budowa dotyczy struktury dokumentu, lecz nie jego zawartości. Z drugiej strony, poprawny dokument XML spełnia kryteria zdefiniowane w swojej definicji DTD lub w schemacie.

DTD i schematy są w istocie dwoma odmiennymi sposobami ustanawiania reguł dotyczących zawartości dokumentu XML. DTD ma dłuższą historię i bardziej ugruntowane standardy, lecz posiada znaczące ograniczenia w porównaniu ze schematem. Po pierwsze, dokument DTD nie może być napisany w XML-u. Oznacza to, iż DTD nie jest dokumentem XML. Po drugie, wybór typów danych dostępnych przy definiowaniu zawartości atrybutu lub elementu jest w DTD bardzo ograniczony. Schemat nie tylko definiuje strukturę danych opisanych przez dokument, lecz również pozwala autorowi definiować określone składniki struktury danych.

Dla jednego dokumentu XML możemy użyć zarówno DTD, jak i schematów, lecz poziom kontroli dostępny w schematach czyni z nich cenniejsze niż DTD narzędzie do zabezpieczania danych i komunikatów definiowanych w dokumencie. Organizacja W3C przedstawiła propozycję standardowej specyfikacji schematu (*http://www.w3.org/XML/Schema.html#dev*).

Schemat jest po prostu zbiorem wstępnie zdefiniowanych reguł, opisujących dane zawarte w dokumencie XML. Ideowo schemat jest bardzo podobny do definicji tablicy w relacyjnej bazie danych. W schemacie XML definiujemy strukturę XML dokumentu, jego elementy, typy danych elementów i skojarzonych z nimi atrybutów, a co najważniejsze, stosunki nadrzędny-podrzędny pomiędzy elementami. Możemy tworzyć schematy na różne sposoby. Jednym z nich jest ręczne wprowadzanie informacji za pomocą Notatnika. Możemy też tworzyć schematy za pomocą narzędzi wizualnych, np. VS.NET i XML Authority. Wiele zautomatyzowanych narzędzi potrafi też generować surowe schematy na podstawie przykładowych dokumentów XML (technika ta przypomina inżynierię wsteczną). Jeśli nie chcemy ręcznie pisać schematu, możemy wygenerować surowy schemat z przykładowego dokumentu XML za pomocą programu VS.NET XML Designer. Następnie będziemy mogli dopasować schemat tak, by stał się ściśle zgodny z naszymi regułami biznesowymi. W VS.NET wygenerowanie schematu z przykładowego dokumentu XML wymaga zaledwie jednego kliknięcia myszą. Aby utworzyć surowy schemat z naszego dokumentu *Catalog1.xml* (patrz: rysunek 4.1 bazujący na listingu z rozdziału 3.):

1. Otwórz plik *Catalog1.xml* (dostępny pod adresem *ftp://ftp.helion.pl/przyklady/hpxmlp.zip*) w projekcie VS.NET. VS.NET wyświetli dokument XML i jego widoki *XML* i *Data* na dole okna.
2. Kliknij *XML* w menu *Main* i wybierz *Create Schema* (Utwórz schemat).



Rysunek 4.1. *Fragment schematu XSD wygenerowanego przez program XML Designer*

I to już wszystko! System utworzy schemat o nazwie *Catalog1.xsd*. Jeśli klikniemy podwójnie ten plik w Solution Explorerze, zobaczymy ekran jak na rysunku 4.1. Proszę zwrócić uwagę na zakładki *DataSet* i *XML* na dole ekranu. Widok *DataSet* omówimy w dalszej części rozdziału.

Na potrzeby naszej dyskusji poniżej zamieściliśmy również listing schematu (*Catalog1.xsd*). Deklaracja schematu XML (*XSD — XML Schema Declaration*) zaczyna się od określonych standardowych wpisów. Wprawdzie kod XSD może wydawać się złożony, lecz nie musimy bać się jego składni. W rzeczywistości strukturalna część XSD jest bardzo prosta. Zgodnie z definicją element może posiadać jedną lub więcej struktur danych `complexType` lub `simpleType`. Struktura danych `complexType` zagnieżdża inne struktury danych `complexType` i `simpleType`. Struktura `simpleType` zawiera jedynie dane.

W naszym przykładzie XSD (patrz poniżej) element `Katalog` może zawierać jeden lub więcej (`unbounded`) egzemplarzy elementu `Produkt`. Wobec tego element `Katalog` jest definiowany jako zawierający strukturę `complexType`. Poza elementami `Produkt` element `Katalog` może zawierać też inne elementy, na przykład `Supplier`. W formancie XSD definiujemy tę regułę za pomocą struktury `choice`, jak poniżej:

```
<xsd:element name="Katalog" msdata:IsDataSet="true">
      <xsd:complexType>
            <xsd:choice maxOccurs="unbounded">
               --- --- ---
               --- --- ---
            </xsd:choice>
      </xsd:complexType>
</xsd:element>
```

Ponieważ element `Produkt` zawiera inne elementy, więc zawiera również strukturę `complexType`. Ta z kolei sekwencję (`sequence`) `NrProduktu` i `CenaSklepowa`. `NrProduktu` i `CenaSklepowa` nie zawierają dalszych elementów, więc w ich definicjach po prostu podajemy ich typ danych. Automatyczny generator nie poradził sobie ze zidentyfikowaniem tekstu w elemencie `CenaSklepowa` jako dziesiętnych danych liczbowych; przekształciliśmy typ danych na dziesiętny ręcznie.

Listing 4.1. *Catalog1.xsd*

```
<xsd:schema id="Katalog"
    targetNamespace="http://tempuri.org/Catalog1.xsd"
    xmlns="http://tempuri.org/Catalog1.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    attributeFormDefault="qualified" elementFormDefault="qualified">
    <xsd:element name="Katalog" msdata:IsDataSet="true"
        msdata:EnforceConstraints="False">
        <xsd:complexType>
            <xsd:choice maxOccurs="unbounded">
                <xsd:element name="Produkt">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="NrProduktu"
                                type="xsd:string" minOccurs="0" />
                            <xsd:element name="NazwaProduktu"
                                type="xsd:string" minOccurs="0" />
                            <xsd:element name="CenaSklepowa"
                                type="xsd:string" minOccurs="0" />
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:choice>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Narzędzia i pułapki...

Kontrola poprawności XML-a w VS.NET

VS.NET udostępnia szereg narzędzi do pracy z dokumentami XML. Jedno z nich pozwala sprawdzić, czy dany dokument XML jest poprawnie zbudowany. Pracując w widoku XML dokumentu XML, możemy z menu *Main* wybrać *XML/Validate XML Data*, aby sprawdzić, czy dokument jest poprawnie zbudowany. System wyświetla zdobyte informacje w lewym dolnym rogu paska stanu. Analogicznie możemy wykorzystać narzędzie *Schema Validation*, aby sprawdzić, czy schemat jest poprawnie zbudowany. W tym celu w widoku XML schematu wybierz z menu *Main* opcję *Schema/Validate Schema*.

Jednakże żaden z powyższych testów nie gwarantuje poprawności danych w XML-u zgodnie z regułami określonymi w schemacie. Aby to sprawdzić, musimy najpierw skojarzyć dokument XML z określonym schematem, co pozwoli sprawdzić poprawność dokumentu. Aby przypisać schemat do dokumentu XML:

1. Wyświetl dokument XML w *widoku XML* (w programie XML Designer).
2. Wyświetl *Property sheet* (arkusz właściwości) — będzie on miał nagłówek *DOCUMENT*.
3. Otwórz rozwijaną listę wyboru po prawej stronie `targetSchema` i wybierz odpowiedni schemat.
4. Teraz możesz sprawdzić poprawność dokumentu XML za pomocą *XML/Validate XML Data* w menu *Main*.

Przy okazji, wiele pakietów oprogramowania innych producentów również potrafi sprawdzać, czy dokument XML jest poprawnie zbudowany oraz jego poprawność według danego schematu. Stwierdziliśmy, iż w tym kontekście bardzo przydatne są programy XML Authority (firmy TIBCO) i XML Writer (Wattle Software). Doskonałe narzędzie o nazwie XSV jest też dostępne pod adresem *http://www.w3.org/2000/09/webdata/xsv*.

Plik XSD jest sam w sobie poprawnie zbudowanym dokumentem XML.

Typy danych w schemacie XSL

Gdy plik XML odgrywa rolę bazy danych, zaś XSL i XPath rolę zapytań SQL tłumaczących plik XML, potrzebujemy miejsca, gdzie zadeklarujemy zawartość pliku XML i związane z nią typy danych. Podobnie jak w każdej bazie danych, nieważne, czy jest to SQL Server czy Oracle, wszystkie kolumny posiadają zdefiniowane typy danych. Rozwiązanie to prowadzi do konieczności wprowadzenia typów danych w schemacie XSL.

Istnieją dwa typy typów danych: proste i pochodne. *Proste typy danych* nie są wyprowadzane z żadnego innego typu danych (np. float — liczby zmiennoprzecinkowe). *Pochodne typy danych* opierają się na innych typach. Typ danych całkowitych (integer), na przykład, pochodzi od danych dziesiętnych (decimal).

Proste typy danych zdefiniowane na potrzeby schematu XML nie muszą być identyczne jak w specyfikacjach innych baz danych, podobnie jak definiowane przez użytkownika typy danych przeznaczone dla schematu XML nie są przeznaczone dla innych zasobów. Tabela 4.4 wymienia różnorodne typy danych, z których mogą korzystać schematy XML.

Wprowadzenie do ataków tekstem jawnym

Ataki tekstem jawnym są jednym z najbardziej podstępnych narzędzi służących hakerom do infiltracji baz danych i aplikacji. Wykorzystują one wykorzystywanie przez XML standardowych znaków języka i fakt, iż znaki te w różnych miejscach aplikacji komputerowej i systemu różne reprezentowane są w różny sposób. Hakerzy wykorzystują niestandardowe kodowanie znaków sterujących (np. końca tekstu lub sterowania przepływem) i łańcuchy pozwalające na dostęp do ukrytych plików, nieosiągalny na inne sposoby, oraz na osadzanie w nich wprowadzanych łańcuchów i komunikatów. Zrozumienie, jak XML interpretuje tekst, jest pierwszym ważnym krokiem w kierunku ochrony baz danych, aplikacji i systemów przed tymi atakami.

Gdy mówimy, że XML jest napisany i komunikuje się tekstem jawnym (zwykłym), mamy na myśli, iż wykorzystuje zestaw znaków ISO-Latin-1. Jest to zestaw znaków używany przez twórców oprogramowania w praktycznie wszystkich krajach Europy Zachodniej i angielskojęzycznych, znany również jako zestaw znaków ASCII (*American Standard Code for Information Interchange*). Inna, bardziej rozszerzona grupa znaków, o ogólnej nazwie *Unicode*, zawiera znaki stosowane w większości znaczących języków świata oraz w matematyce, logice i przy rysowaniu prostych obiektów. Zestaw znaków Unicode jest odwzorowany bezpośrednio na ISO-Latin-1, zaś oba zestawy znaków dają dostęp do liter, cyfr, znaków przestankowych oraz interesujących

Tabela 4.4. *Typy danych w schemacie XML*

Podstawowy typ danych	Pochodny typ danych	Podstawowe aspekty	Ograniczenia
String	normalizedString	equal	length
Boolean	Token	ordered	minLength
Decimal	Language	bounded	maxLength
Float	NMTOKEN	cardinality	pattern
Double	NMTOKENS	numeric	enumeration
Duration	Name	n/d	whiteSpace
dateTime	NCName	n/d	maxInclusive
Time	ID	n/d	maxExclusive
Date	IDREF	n/d	minExclusive
gYearMonth	IDREFS	n/d	minInclusive
gMonthDay	ENTITY	n/d	totalDigits
GDay	ENTITIES	n/d	fractionDigits
GMonth	Integer	n/d	n/d
hexBinary	nonPositiveInteger	n/d	n/d
base64Binary	negativeInteger	n/d	n/d
AnyURI	Long	n/d	n/d
Qname	Int	n/d	n/d
NOTATION	short	n/d	n/d
GYear	Byte	n/d	n/d
n/d	nonNegativeInteger	n/d	n/d
n/d	unsignedLong	n/d	n/d
n/d	unsignedInt	n/d	n/d
n/d	unsignedShort	n/d	n/d
n/d	unsignedByte	n/d	n/d
n/d	positiveInteger	n/d	n/d

znaków dodatkowych (na przykład, sterujących przepływem informacji przez aplikacje i wskazujących systemowi, czy łańcuch wejściowy został z powodzeniem odebrany). Dodatkowe informacje o możliwościach i atrybutach Unicode przedstawimy w dalszej części rozdziału.

Bezpośrednie manipulowanie zestawem znaków wymaga umieszczenia liczbowej reprezentacji znaku pomiędzy symbolami (&) i średnika (;). Ta konwencja jest nieco odmienna od stosowanej w HTML-u, gdzie wartość liczbowa jest umieszczana pomiędzy zestawem znaków &# a średnikiem. Na przykład,

```
&65;
```

oznacza literę A. Cyfrę 2 w XML-u przedstawimy, używając:

```
&50;
```

Te przykłady są proste i oczywiste. Z drugiej strony, poniższy kod jest tłumaczony na „anuluj wiersz":

```
&24;
```

Nagle możliwości ASCII zaczynają się wydawać nieco większe.

Zarówno w HTML-u, jak i w XML-u znaki mogą być przekazywane jako element łańcucha wejściowego lub komunikatu na jeden z trzech sposobów. Do każdego drukowalnego znaku używanego przez XML możemy odwołać się na trzy sposoby: poprzez symbol (do którego jesteśmy przyzwyczajeni), nazwę i przez kod szesnastkowy. Najpowszechniejszym sposobem jest po prostu wpisanie znaku — na przykład symbol „mniejszy niż" jest zapisywany jako <. Do znaku tego możemy też odwołać się przez jego nazwę, jeśli poprzedzimy ją znakiem &. W tym przypadku „mniejszy niż" („*less than*") zapiszemy:

```
&lt
```

Trzecia metoda, najczęściej stosowana przez hakerów do przeprowadzenia ataku tekstem jawnym, polega na wprowadzeniu reprezentacji dziesiętnej znaku. XML wymaga zamknięcia tej liczby pomiędzy symbolami & i ;. W tej metodzie „mniejszy niż" zapiszemy:

```
&60;
```

Ten zapis różni się nieco od stosowanego w HTML-u, gdzie reprezentacja liczbowa jest zamykana pomiędzy &# a średnikiem, więc symbol „mniejszy niż" jest zapisywany:

```
&#60;
```

Niektóre znaki w zestawach znaków używanych przez większość aplikacji posiadają jedynie dwie reprezentacje: nazwę i kod liczbowy, ponieważ są znakami niedrukowalnymi — sterującymi. Znaków sterujących jest cała grupa, od znaku powrotu karetki (&13;) i spacji (&32;) zaczynając, a kończąc na znakach „koniec transmisji" (&4;) i „potwierdzenie negatywne" (&21;). Osadzanie znaków sterujących aplikacją lub systemem w strumieniu otwartego tekstu zwiększa użyteczność zestawów znaków, lecz zarazem zwiększa podatność na ataki.

Ataki tekstem jawnym

Programiści i twórcy baz danych najczęściej korzystają z reprezentacji liczbowej znaków ASCII do pracy ze znakami nieobecnymi na standardowej angielskiej klawiaturze. Na przykład, znaki spotykane w nazwach nordyckich i akcentowane znaki obecne w wielu słowach francuskich, hiszpańskich i niemieckich możemy łatwo wyrazić poprzez reprezentacje liczbowe. Nawet jeśli baza danych i jej interfejs używają jedynie języka angielskiego, reprezentacje numeryczne pozwalają na pewien poziom kontroli typograficznej przekraczający możliwości standardowej klawiatury. Na przykład, określone typy spacji (o różnej długości) i myślników są definiowane i dostępne w pełnym zestawie znaków ASCII, chociaż nie znajdziemy ich na standardowych klawiaturach komputerowych.

Włamanie do Internet Information Services (IIS) Microsoftu poprzez niekanoniczny łańcuch wejściowy jest tylko jednym z wielu przykładów ataków tekstem jawnym. Na stronie WWW Computer Emergency Response Team (CERT) znajdziemy niemal 30 niezależnych ostrzeżeń o miejscach podatnych na atak tekstem jawnym, zaś przeszukując Internet, znajdziemy kolejne setki przykładów. Ostrzeżenia publikowane przez CERT i inne serwisy WWW wskazują, iż niekanoniczne kodowanie znaków nie jest jedynym narzędziem, jakiego hakerzy mogą użyć do infiltracji aplikacji. Czasami sama objętość jawnego tekstu wystarcza, by przysporzyć problemów atakowanym systemom i aplikacjom.

Wiele ataków tekstem jawnym wykorzystuje takie luki w zabezpieczeniach, jak np. bufory wejściowe aplikacji, które mogą ulec przepełnieniu i przekazać dane bezpośrednio do strumieni wykonywanych, zamiast przez normalne zabezpieczające analizatory składniowe. Ograniczenia długości łańcuchów wejściowych w aplikacjach są ważnymi narzędziami pomagającymi ograniczyć dostęp hakerów do tych najczęściej stosowanych metod włamań.

Pełny zestaw znaków ASCII składa się z 256 odrębnych jednostek. Większość z nich stanowią litery, cyfry i inne znaki drukowalne, lecz dwa zakresy definicji nie klasyfikują się jako normalne definicje znaków. Znaki o numerach od 0 do 31 są rozkazami dla drukowania lub urządzeń komunikacyjnych. Ich zakres rozciąga się od powrotu karetki (Carriage Return — `&13;`) aż do Device Control 3, znaku ogólnie zarezerwowanego dla komunikatu XOFF (`&19;`). Znaki od 128 do 159 nie są zdefiniowane w standardzie i zostały zarezerwowane na przyszłe potrzeby lub dla indywidualnych implementacji. Oznacza to, że działanie znaków z tego zakresu jest zależne od przeglądarki, bazy danych lub innych aplikacji, które interpretują dokument. W najlepszym przypadku, jeśli definicja znaku nie została z góry ustalona, nie zdefiniowany znak będzie po prostu ignorowany. W najgorszym razie reakcja aplikacji może być nieprzewidywalna.

Przykład: kody ucieczki HTML

Dane przenoszone w znakowych komunikatach XML-a mogą zawierać znaki kodowane w ASCII i Unicode, nazwy znaków XML i reprezentacje liczbowe oraz szesnastkowe reprezentacje kodów znaków i ucieczki języka HTML. Kody ucieczki HTML stanowią ciekawą lukę w zabezpieczeniach, ponieważ bardzo rzadko uznawane są za niebezpieczne, a jednak możliwości zaszkodzenia za ich pomocą są olbrzymie.

Jak można użyć zestawu znaków do ataku? Wiele luk w zabezpieczeniach ma związek z nieautoryzowanymi zmianami informacji wyświetlanych na ekranie. Weźmy na przykład stronę WWW zawierającą komunikat, który, aby był widoczny w jak największej liczbie przeglądarek, używa kolorów nazwanych w języku HTML 4.0. Twórca strony bezpośrednio definiuje kolory: czarny (`#000000`) dla tekstów i żółty (`#FFFF00`) dla tła. Jeśli wstawimy do tekstu znaczniki definiujące określone fragmenty tekstu jako żółte, wówczas ten tekst będzie obecny, lecz niewidoczny dla użytkowników odwiedzających stronę. W tym przypadku prosta kontrola, czy każde odwołanie do danych posiada własny wpis, nie wykaże żadnych problemów, podobnie proste sprawdzenie wygenerowanego kodu źródłowego strony również może nie wskazać ich istnienia.

Inny przykład dotyczy znaków nie wyświetlanych. Należą do nich znaki sterujące takie jak Escape (`&27;`) i typograficzne, np. spacja (`&32;`). Luki w bezpieczeństwie

związane z tymi znakami biorą się z faktu, iż reprezentacja ASCII pozostaje niezmienna, zaś inne (np. zestaw Unicode, który omówimy za chwilę) mogą być różne w różnych językach.

Jeden z godnych uwagi exploitów poprzez przepełnienie bufora tekstem jawnym dotyczył Oracle 9i i luki w bezpieczeństwie przekazywanej do tego oprogramowania przez Apache — program typu open source, którego Oracle używa jako serwera WWW dla swojego mechanizmu bazy danych. Apache Procedural Language/Structured Query Language jest modułem instrukcji używanym przez Oracle. Okazało się, że prosty atak tekstem jawnym wykorzystujący łańcuchy dłuższe niż oczekiwane przez aplikację, mógł spowodować przepełnienie bufora, co pozwalało na parsowanie i przesłanie tekstu, który przepełnił bufor, bez interwencji standardowego kodu zabezpieczeń. Zapisane w bazie danych procedury mogły być wykonane z uprawnieniami serwera Apache. Biorąc pod uwagę, iż serwer Apache w systemach opartych na Windows NT zwykle działa na poziomie systemu, haker wykorzystujący to podejście mógł zyskać pełną kontrolę nad atakowanym systemem.

Firma Oracle wypuściła łatę, zaproponowano też techniki obejścia tej luki, lecz ataki tego typu są najczęściej spotykane i prędzej czy później uderzają we wszelkich producentów większych aplikacji, systemów operacyjnych i routerów sieciowych.

Unicode

Unicode jest szerokim zbiorem standardów dotyczących zestawów znaków używanych przez większość ważniejszych języków na świecie. Dla twórcy dokumentów XML elastyczność, jaką oferuje Unicode, jest istotna, lecz różnorodne sposoby, na jakie aplikacje mogą interpretować informacje niesione przez znaki Unicode powodują nieuniknione powstawanie miejsc podatnych na ataki. Pełną listę zestawów znaków Unicode i sposoby ich użycia znajdziemy pod adresem *http://www.unicode.org*.

Gdy system używa jednocześnie zestawów znaków ASCII i Unicode, mogą pojawić się pewne problemy wynikające z podstawowych różnic w obu standardach kodowania. Tradycyjny zestaw ASCII stosuje kodowanie przy wykorzystaniu 8 bitów, przez które liczba znaków w standardzie jest ograniczona do 256. Ponieważ praktycznie wszystkie systemy operacyjne używają ASCII do kodowania informacji wyświetlanych i drukowanych, zaś Unicode stosują jako rozszerzony kod odwzorowany na ASCII, więc procedury zabezpieczeń szukające określonych „zakazanych" łańcuchów znaków ASCII mogą przepuścić potencjalnie szkodliwe instrukcje osadzone w adresie URL.

Jedna z głównych luk w zabezpieczeniach wynika z wyboru metody odwzorowania Unicode na ASCII. Ponieważ Unicode musi radzić sobie z wieloma różnymi symbolami w wielu językach, znaki mogą mieć długość 16, 24, a nawet 32 bitów. Wszystkie znaki łacińskie (używane w języku angielskim) są 16-bitowe, lecz część z tych znaków (w tym przestankowe i sterujące) możemy znaleźć też w innych językach. W tych innych zestawach znaków ukośniki, kropki i znaki sterujące mogą mieć dłuższe reprezentacje niż w zestawie znaków łacińskich.

Unicode Consortium definiuje metody odwzorowania w UTF-8 (*Unicode Transformation Format-8*). UTF-8 podaje, iż wszelkie oprogramowanie kodujące dane do Unicode musi używać najkrótszej z możliwych implementacji. Standard pozostawia jednak

otwartą możliwość użycia przez program dowolnej możliwej reprezentacji przy dekodowaniu znaków. Ta niejednoznaczność może zostać wykorzystana do przepuszczenia przez proces zabezpieczający znaków zakazanych.

W powszechnie znanym incydencie serwer IIS Microsoftu stał się podatny na żądanie dostępu do bezpiecznych plików. Standardowo łańcuch typu:

```
/../../
```

nie jest przepuszczany przez procedury bezpieczeństwa IIS, ponieważ poprzez adresowanie względne mógłby dać dostęp do katalogów. Gdy do adresu URL został w poniższej sekwencji wstawiony ciąg Unicode `%c0%af`, wówczas procedury kontrolne zabezpieczeń, zaprogramowane do dekodowania najkrótszej implementacji, nie rozpoznały w nim ukośnika (/).

```
%c0%af../..%c0%af
```

Łańcuch był przekazywany do interpretera poleceń, który dekodował go w bardziej elastyczny sposób jako adres względny, co pozwalało wykorzystać lukę w zabezpieczeniu.

Sposoby kontroli poprawności XML-a

Sprawdzanie poprawności XML-a jest formalnym procesem kontroli zgodności plików XML z odpowiednimi DTD, schematami lub jednym i drugim. Najpierw musimy jednak wyraźnie powiedzieć, że dokument XML do funkcjonowania nie wymaga DTD ani schematu. Dokument nie może zostać *uznany* za poprawny, o ile nie posiada odnośnika do przynajmniej jednego z dwóch powyższych dokumentów i jeśli poprawność tego odwołania nie została sprawdzona przez odpowiedni procesor (program do kontroli poprawności). Musimy wiedzieć, w jakiej kolejności DTD i schematy są stosowane przy kontroli poprawności dokumentu XML oraz co dokładnie jest kontrolowane, by móc prawidłowo wykorzystać wbudowane możliwości procesorów XML-a dla zapewnienia bezpieczeństwa. Musimy też wiedzieć, czego te mechanizmy *nie* robią, aby utworzyć odpowiednie procedury wewnętrznej kontroli poprawności danych przekazywanych przez XML.

Mechanizmy kontroli poprawności XML-a, zarówno DTD, jak i schematy, mają przede wszystkim na celu zachowanie jakości struktur, ograniczeń typów danych i wymuszanie jednolitości w obrębie organizacji lub systemu aplikacji. Nie zostały one zaprojektowane ani nie nadają się zbytnio do kontroli spójności danych i ich poprawności dla danej aplikacji. Jeśli wyobrazimy sobie te dwa mechanizmy kontroli poprawności jako sita, wówczas formalną kontrolę poprawności XML-a możemy uznać za sito o dużych oczkach, które odsiewa poważne niespójności struktury i danych. Drobniejszym sitem, które zapewnia utrzymanie danych w granicach rozsądku (na przykład, aby cena gumy do żucia nie wynosiła 50 zł zamiast 0,50 zł), będą procedury weryfikacji danych pisane przez lokalnego programistę. W tych procedurach dane wejściowe muszą być kontrolowane pod względem poprawności, prawidłowo dekodowane, a następnie ich zawartość weryfikowana. Wszystko to musi odbywać się w sposób nie obciążający serwera lub oprogramowanie klienta w niedopuszczalnym stopniu.

Zyski z właściwej kontroli poprawności są olbrzymie. Po pierwsze, dobra kontrola poprawności i weryfikacja uniemożliwiają przeprowadzenie większości popularnych typów ataków tekstem jawnym, jakie omawialiśmy do tej pory. Znaki kodowane w nietypowy sposób lub o zdekodowanej wartości wykraczającej poza granice logicznych parametrów danych są filtrowane ze strumienia danych, zanim zostaną wykonane lub zapisane w bazie danych. Ponadto kontrola jakości danych jest lepsza, ponieważ wpisy wykraczające poza granice logiczne są odrzucane na etapie wejścia.

Kontrola poprawności wprowadzanego tekstu

Bardzo silna może być pokusa, by zdecydować, iż istniejące mechanizmy kontroli poprawności XML-a powinny całkowicie wystarczyć do zabezpieczania danych wprowadzanych przez dokumenty XML. Jak, niestety, przekonaliśmy się, hakerzy zbyt łatwo mogą wykorzystać różnice pomiędzy zestawami znaków tekstu jawnego do zaatakowania systemu używającego poprawnie zbudowanych i poprawnych dokumentów XML. Wobec tego twórca aplikacji musi zbudować niezależne procedury kontroli poprawności danych, przechodzących do aplikacji przez dokument XML sprawdzony pod względem poprawności.

Odpowiednie podejście polega na rozbiciu problemu weryfikacji na szereg odrębnych kroków. Pierwszym w kolejności (aczkolwiek omówimy go na końcu) jest formalna kontrola poprawności dokumentów definiujących podstawowe dane poprzez analizatory składni DTD i schematu. Następnie odbywa się obróbka potoku wejściowego przy odbiorze przez aplikację. Kolejnym ważnym krokiem jest upewnienie się, czy każdy znak wejściowy jest poprawny w ramach definicji języka oraz czy każdy został zdekodowany zgodnie z odwzorowaniem uznanym przez wszystkie składniki aplikacji. Na koniec zmuszenie, by każdy prawidłowo zdekodowany wpis mieścił się w logicznych granicach aplikacji, pomaga w wyeliminowaniu zarówno celowo wprowadzonego złośliwego kodu, jak i niezamierzonych konsekwencji pomyłek ludzkich.

Sprowadzenie do postaci kanonicznej

Sprowadzenie do postaci kanonicznej oznacza zdolność do nadania dokumentowi najprostszej możliwej formy. Proces ten tworzy dokumenty równe sobie semantycznie z nierównych poprzez normalizację danych, analizę składniową i aranżację elementów w formę neutralną składniowo. Sprowadzanie do postaci kanonicznej omówimy nieco bardziej szczegółowo w rozdziale 6., lecz obecnie musimy pokrótce opisać zastosowanie tego procesu w podpisach cyfrowych XML.

Sprowadzanie do postaci kanonicznej w podpisach cyfrowych XML

Natura Unicode powoduje, iż niektóre najczęściej używane znaki (spacje, powrót karetki, koniec wiersza itp.) są reprezentowane w zestawach znaków o różnych długościach. W najnowszych wersjach standardów kodowania organizacja Unicode nakazała, by wszelkie oprogramowanie kodowało znaki do najkrótszych reprezentacji, jednakże oprogramowanie ma prawo dekodować wszelkie możliwe reprezentacje, aby utrzymać

zgodność wstecz ze starszymi wersjami programów. Oznacza to, że istniejące analizatory składni XML sprowadzają kilka różnych reprezentacji szesnastkowych do tych samych znaków, co może stworzyć możliwości ataków wspomnianych wcześniej w niniejszym rozdziale.

Twórcy dokumentów XML muszą też zdawać sobie sprawę z różnych poziomów obsługi kodów ASCII i Unicode w używanych przez siebie narzędziach programistycznych. Języki programowania takie jak *Perl*, *Python* czy *Tcl* oraz interfejsy typu *Simple API for XML (SAX)* i *Document Object Module (DOM)* są powszechnie stosowane przy programowaniu mającym związek z XML-em. Każdy z nich obsługuje jedną lub kilka odmian znaków Unicode, lecz poszczególne języki i interfejsy różnią się zdecydowanie sposobami działania tej obsługi.

Na przykład, Perl zwraca dane w formacie UTF-8, mimo że nie obsługuje pełnych implementacji Unicode. Jeśli potrzebne są znaki spoza UTF-8, wówczas muszą być bezpośrednio obsługiwane przez moduł `Unicode::String`. Niektóre procesory XML-a dostępne dla Perla, np. SAX i DOM, obsługują pełny standard Unicode. Ponieważ procesorów SAX i DOM jest kilka, a każdy z nich traktuje Unicode w nieco odmienny sposób, radzimy przejrzeć dokumentację używanego modułu, aby sprawdzić szczegóły kodowania znaków.

W przeciwieństwie do języka Perl, Python nie używa Unicode ani żadnej jego formy jako wewnętrznego formatu kodowania znaków. Zamiast tego udostępnia łańcuchy Unicode jako dostępny dla programisty typ obiektu danych. Każdy łańcuch znakowy może zostać zakodowany jako obiekt Unicode, jeśli umieścimy przed łańcuchem znak `u`. Na przykład:

```
fastship = 'Wysylka tego samego dnia'
```

Powyższy łańcuch zostanie zakodowany w ASCII. Następny przykład koduje ten sam łańcuch w Unicode:

```
fastship = u'Wysylka tego samego dnia'
```

Tcl obsługuje Unicode bezpośrednio poprzez analizator składni TclXML. Jeśli chcemy obrabiać określony łańcuch znaków zakodowany w inny sposób, wówczas funkcja encoding pozwala na łatwe przejście z Unicode na ASCII, z UTF-8 na UTF-16 oraz pomiędzy dowolnymi innymi metodami kodowania znaków obsługiwanymi przez określony system.

Powinno być oczywiste, że przy tak wielkiej liczbie sposobów kodowania danych znakowych zależnych od użytych narzędzi programistycznych spoczywa na programiście obowiązek opracowania procedur kontroli poprawności danych wprowadzanych do aplikacji poprzez XML. Aby ustrzec się ataków opartych na dłuższych niż minimalne reprezentacjach znaków, może okazać się konieczna obsługa wielu zestawów znaków Unicode poprzez bezpośrednie instrukcje w odpowiednich procesach. Zamiast tego możemy też zdecydować o obsłudze jedynie kodowania w UTF-8, zwłaszcza jeśli lista języków używanych w zbiorach danych dla aplikacji jest ograniczona.

Narzędzia i pułapki

Narzędzia kontroli poprawności dokumentów XML

Narzędzia służące do kontroli poprawności dokumentów XML i związanych z nimi danych możemy zaklasyfikować zgodnie z trzema podstawowymi etapami kontroli poprawności, niezbędnymi do zminimalizowania możliwości wystąpienia niezamierzonych lub złośliwych szkód w systemie. Tymi trzema etapami są integralność XML-a, sprowadzenie danych wejściowych do postaci kanonicznej i kontrola poprawności w aplikacji. Dla dwóch pierwszych etapów dostępne są narzędzia pozwalające tworzyć aplikacje odporne na ataki. We wszystkich trzech obszarach musimy znaleźć odpowiednią równowagę pomiędzy siłą metod kontroli poprawności a kosztem i potencjalnymi słabościami. Przyjrzyjmy się każdemu etapowi osobno:

- **Integralność XML-a** — dokumenty XML poprawnie zbudowane i poprawne stanowią podstawę dla poprawnych danych. Zarówno DTD, jak i schematy są bardzo pomocne przy tworzeniu właściwych dokumentów, zaś do zapewnienia zgodności dokumentu ze standardami XML, DTD i schematami dostępne są odpowiednie narzędzia. O'Reilly, Brown University i W3C Consortium udostępniają narzędzia online, pozwalające na skanowanie i kontrolę poprawności dokumentów XML. Każde z tych narzędzi jest inne; Brown University oferuje najbardziej szczegółowe raporty, zaś O'Reilly najbardziej zwięzłe, lecz stosując dwa z nich lub więcej możemy upewnić się, czy nasz kod jest prawidłowo zbudowany i zgodny z załączonymi DTD i schematami. Narzędzia uruchamiane lokalnie również są dostępne. XML Spy firmy Altova jest wiodącym komercyjnym narzędziem służącym do tworzenia i kontroli poprawności dokumentów XML. Microsoft i Sun Microsystems udostępniają narzędzia kontroli poprawności do darmowego ściągnięcia, aczkolwiek narzędzie Microsoftu nie jest wspierane przez producenta, zaś często aktualizowany produkt firmy Sun jest podstawą dla narzędzia kontroli poprawności XML-a w oprogramowaniu open source Apache.
- **Dane wejściowe w postaci kanonicznej** — istnieje wiele sposobów, na jakie znaki, zwłaszcza niedrukowalne i wspólne dla wszystkich języków, mogą być reprezentowane liczbowo w systemie. Właściwe stosowanie funkcji języków programowania, na przykład `MultiByteWideToChar` Microsoftu, `encode()` i `unicodedata` z języka Python oraz `convertfrom` i `convertto` z języka Tcl pozwala zapewnić konwersję znaków z wielu różnych zestawów Unicode do wspólnej, najkrótszej reprezentacji, zanim rozpocznie się przetwarzanie zakodowanych danych.
- **Kontrola poprawności dla aplikacji** — ostatnim krokiem przy zapewnianiu integralności danych jest zagwarantowanie, by wszystkie dane wejściowe posiadały odpowiedni typ, konfigurację i zakres na potrzeby danej aplikacji. Dla tego procesu nie istnieją żadne „standardowe" narzędzia, ponieważ aplikacje różnią się między sobą, lecz twórcy oprogramowania muszą znać pewne zasady:

  1. Schematy nie najlepiej nadają się do kontroli poprawności danych wejściowych w intensywnie używanych serwisach, ponieważ muszą być wywoływane i interpretowane dla każdego wprowadzenia danych.
  2. Java jest dobrym narzędziem kontroli poprawności danych, ponieważ może pobrać schemat i na jego podstawie opracować model dla narzędzia kontrolującego poprawność.
  3. Wszelka kontrola danych wejściowych jest kosztowna, ponieważ zajmuje zasoby systemowe i czas procesora. Popularna aplikacja może wywoływać program kontrolujący poprawność danych setki lub tysiące razy na minutę. Optymalizacja kodu jest tu niezbędna.

Kontrola poprawności Unicode

Ochrona systemu przed atakami jawnym tekstem polega głównie na kontroli kodowania stosowanej przy przekazywaniu znaków z jednego formatu do innego, zazwyczaj pomiędzy ASCII a jedną z postaci Unicode. Jako przykład sposobów, na jakie systemy obsługują konwersję, przytoczymy funkcjonalność konwersji wejściowych łańcuchów znakowych na Unicode udostępnianą przez Microsoft.

Funkcja `MultiByteToWideChar` odwzorowuje wejściowy łańcuch znaków na łańcuch Unicode multibyte (wielobajtowy) niezależnie od tego, czy wprowadzane znaki wymagają reprezentacji wielobajtowej czy nie. Z punktu widzenia kosztów funkcja zyskuje pojedyncze, jednorodne odwzorowanie wszystkich znaków kosztem pamięci. Ponieważ ataki tekstem jawnym zwykle wykorzystują różnice w reprezentacji charakterów kodowanych na ośmiu i szesnastu bitach, wyrównanie ustawień dla wszystkich łańcuchów wejściowych jest dobrym wstępem do eliminacji problemu. Struktura i argumenty `MultiByteToWideChar` są następujące:

Listing 4.2. *Struktura i argumenty MultiByteToWideChar*

```
Int MultiByteToWideChar {
UINT CodePage,
DWORD dwFlags,
LPCSTR lpMultiByteStr,
Int cbMultiByte,
LPWSTR lpWideCharStr,
Int cchWideChar
};
```

CodePage oznacza tutaj stronę kodową używaną przy konwersji. W chwili obecnej obsługiwane są dwie wartości. Przyjrzyjmy się bardziej szczegółowo tej funkcji:

- `CP_ACP` — używa strony kodowej ANSI.
- `CP_OEMCP` — odnosi się do strony kodowej OEM.
- `dwFlags` — ustala, czy znaki są zestawieniem, czy mają wartości proste (`MB_PRECOMPOSED`) czy są złożone (`MB_COMPOSITE`); czy znaki powinny zostać przetłumaczone na kształty, a nie na znaki sterujące (`MB_USEGLYPHCARS`), oraz czy komunikat o błędzie powinien zostać zgłoszony przy napotkaniu niepoprawnego znaku (`MB_ERR_INVALID_CHARS`).
- `lpMultiByteStr` — wskazuje na wejściowy łańcuch znaków.
- `cbMultiByte` — rozmiar łańcucha wejściowego w bajtach. Jeśli parametr ma wartość 1, wówczas długość obliczana jest automatycznie.
- `lpWideCharStr` — bufor wyjściowy, w którym zostanie umieszczony łańcuch po translacji.
- `cchWideCar` — rozmiar bufora wyjściowego w znakach wide character.

Poza funkcją `MultiByteToWideChar` Microsoft obsługuje translację znaków poprzez skojarzone funkcje, na przykład `WideCharToMultiByte`, która zasadniczo odwraca proces

`MultiByteToWideChar`, `TranslateCharsetInfo`, która dokonuje translacji na podstawie określonego zestawu znaków i przydaje się, gdy napotkamy znane języki oparte na innym zestawie znaków niż łaciński, oraz `IsDBCSLeadByte`, która ustala, czy dany znak powinien zostać przetłumaczony jako znak jednobajtowy, czy też jako pierwszy bajt dwubajtowego znaku złożonego.

Inne języki umożliwiają na zbliżonym poziomie kontrolę nad translacją pomiędzy łańcuchami ASCII i Unicode. Widzieliśmy już, na przykład, że Python pozwala na zakodowanie łańcucha wejściowego w Unicode poprzez dodanie prefiksu `u`. Python pozwala również na bardziej precyzyjną kontrolę poprzez inne dostępne funkcje. Kodowanie w określonym zestawie językowym możemy wyspecyfikować za pomocą metody `encode()`. Weźmy poprzedni przykład:

```
fastship = u'Wysylka tego samego dnia'
```

Możemy wymusić konwersję tego łańcucha z być może nieznanego formatu Unicode, właściwego dla systemu hosta, na format Unicode odpowiadający bezpośrednio tablicy 8-bitowych znaków ASCII w sposób następujący:

```
fastship = 'Wysylka tego samego dnia'
fastship.encode ('latin-1')
```

Podobne metody mogą posłużyć do translacji z formatu Unicode na inny. Translacja z innego formatu do Unicode wykorzystuje format `unicode()`, który może przyjmować argumenty wymuszające określony kod Unicode. Na przykład:

```
unicode ('Wysylka tego samego dnia','utf-16')
```

Ten wycinek programu koduje łańcuch w zestawie znaków dwubajtowych Unicode UTF-16.

Podczas takich translacji możliwa jest próba translacji znaku do zestawu, w którym znak ten nie jest reprezentowany. Python udostępnia trzy sposoby obsługi tego błędu wybierane przez programistę. Opcje te używane są w metodzie następująco:

```
unicode ('Wysylka tego samego dnia','utf-16','strict|ignore|replace')
```

Podanie `strict` spowoduje niepowodzenie metody, jeśli odwzorowanie będzie niemożliwe; `ignore` prowadzi do usunięcia nie przekształconego znaku z łańcucha wyjściowego, zaś `replace` zastępuje problematyczny znak przez `\uFFFD` (oficjalny znak zastępczy w języku Python). W każdym z dostępnych kodów znak `\uFFFD` będzie definiowany indywidualnie.

W przedstawionych przykładach system programistyczny udostępnia programiście narzędzia, pozwalające kontrolować sposób translacji. Niezależnie od używanego systemu najważniejszą czynnością programisty będzie aktywny wybór i konsekwentne stosowanie metody translacji w dokumencie XML i w aplikacji. Konsekwentne stosowanie jednego schematu translacji minimalizuje prawdopodobieństwo niezamierzonych konsekwencji i nieumyślnych luk w zabezpieczeniach powodowanych przez niedopasowane reprezentacje znaków przy przekazywaniu danych z jednego komponentu oprogramowania do innego.

Kontrola poprawności dokumentu lub komunikatu

Gdy już w rozsądnych granicach możemy założyć, że dane wprowadzane do systemu składają się z legalnych i nieszkodliwych znaków, wówczas zaczyna się etap procesu kontroli poprawności, który pochłania najwięcej zasobów komputera. Na tym etapie kontrolowane są dane i komunikaty w celu upewnienia się, czy wartości prawidłowo mieszczą się w określonych dla danej aplikacji granicach.

Pomyślmy o aplikacji katalogu, do której odwoływaliśmy się w tym rozdziale. Na wartości można przypuszczalnie nałożyć kilka typów ograniczeń, aby zagwarantować, by dane były odpowiednie dla aplikacji. Na przykład, ceny będą danymi liczbowymi, a nie znakami alfabetu. Numery kart kredytowych użytych do zapłaty będą mieścić się w znanym zakresie długości, podobnie jak numery telefonów i kody pocztowe. Każda z tych wartości jest kandydatem do ścisłej kontroli poprawności.

Ponieważ pracujemy z XML-em, możemy wykorzystać schemat jako narzędzie do ograniczania danych. W przypadku numerów telefonicznych łatwo jest zbudować elementy schematu zapewniające, by numer był zgodny z podstawowym systemem (przykład dotyczy numerów w USA). Fragment schematu dla aplikacji katalogu został przedstawiony poniżej. Fragment ten dopuszcza jedynie dane zgodne z formatem standardowych numerów telefonów w USA, aczkolwiek pozwala też na stosowanie maksymalnie pięciocyfrowych numerów wewnętrznych.

Listing 4.3. *Fragment schematu, ograniczający dane do postaci standardowych numerów telefonicznych USA*

```
<xsd:schema id="Catalog"
    targetNamespace="http://tempuri.org/Catalog1.xsd"
    xmlns="http://tempuri.org/Catalog1.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    attributeFormDefault="qualified" elementFormDefault="qualified">
    <xsd:complexType name="telephone">
       <xsd:sequence>
          <xsd:element name="areacode">
              <xsd:simpleType>
                 <xsd:restriction base="xsd:string">
                    <xsd:pattern value="\d\d\d"/>
                 </xsd:restriction>
              </xsd:simpleType>
          </xsd:element>
          <xsd:element name="exchange">
              <xsd:simpleType>
                 <xsd:restriction base="xsd:string">
                    <xsd:pattern value="\d\d\d"/>
                 </xsd:restriction>
              </xsd:simpleType>
          </xsd:element>
          <xsd:element name="number">
              <xsd:simpleType>
                 <xsd:restriction base="xsd:string">
                    <xsd:pattern value="\d\d\d\d"/>
                 </xsd:restriction>
```

```
            </xsd:simpleType>
        </xsd:element>
        <xsd:element name="extension">
            <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d\d\d\d\d"/>
               </xsd:restriction>
            </xsd:simpleType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
```

Ten fragment, wykorzystujący dopasowanie do wzorca w celu wymuszenia sekwencji cyfr XXX-XXX-XXXX-XXXXX (dopuszczającą pięciocyfrowy numer wewnętrzny), jest łatwy do napisania i zrozumienia. Proces pisania schematu nakładającego ograniczenia na wszystkie dane katalogowe byłby stosunkowo prosty, aczkolwiek nieco żmudny. Wiedząc, że jesteśmy w stanie to uczynić, możemy przemyśleć, czy naprawdę tego chcemy.

Mimo dużych możliwości schematy XML trapi kilka problemów, z powodu których raczej nie warto używać ich jako jedynego mechanizmu kontroli poprawności danych dla naszej aplikacji. Po pierwsze, specyfikacja schematu XML nie została jeszcze ukończona i zatwierdzona. Jeśli zbudujemy kod aplikacji oparty na schemacie, istnieje realne prawdopodobieństwo, iż drobne zmiany w ostatecznej specyfikacji będą mogły spowodować konieczność zmian kodu aplikacji. Drugi problem jest chyba poważniejszy, ponieważ dotyczy bezpośrednio szybkości działania aplikacji.

Schemat XML musi być wywoływany i parsowany za każdym razem, gdy zaistnieje potrzeba kontroli poprawności. W przypadku cieszącej się popularnością aplikacji katalogu lub informacji telefonicznej może to oznaczać tysiące żądań kontroli poprawności danych na minutę. Możliwa jest konwersja schematu na obiekt dokumentu DOM i zapisanie w pamięci podręcznej, co zaoszczędzi czasu wymaganego do pobrania schematu z dysku przy każdym wywołaniu, lecz wciąż nie pozbędziemy się w ten sposób konieczności analizy składni schematu przy każdym wywołaniu. Gdy spojrzymy na obciążenie systemu komputerowego przez powtarzające się przebiegi analizy składni schematu, stanie się oczywiste, iż kontrola poprawności danych oparta jedynie na modelu schematu jest bardzo, bardzo kosztownym rozwiązaniem.

Znacznie rozsądniejszą metodą jest zastosowanie schematu do zdefiniowania ograniczeń danych, a następnie konwersja schematu na program w języku np. Tcl, Python lub Java. Taka metoda kontroli poprawności również obciąża system, lecz w mniejszym stopniu niż poprzednia. Proces musi być jak najszybszy, co oznacza, że w tym miejscu powinien być stosowany najwydajniejszy sprzęt, aby nie stał się wąskim gardłem. Tutaj też można użyć serwerów równoległych, rozkładając obciążenie obliczeniowe na kilka fizycznych komputerów.

Czy XML jest poprawnie zbudowany?

Pierwszym krokiem ochrony aplikacji przed atakami (lub nawet przed niezamierzonymi poważnymi błędami prowadzącymi do katastrofy) jest zapewnienie, by struktura

i zawartość strumienia danych zachowywały się zgodnie z oczekiwaniami programisty. XML udostępnia do tego celu dwa narzędzia: DTD i schematy. Jak już pokazaliśmy, DTD są narzędziami weryfikującymi poprawną budowę dokumentu XML, to znaczy zgodność z właściwą gramatyką i sposobem użycia XML-a, oraz konsekwentne użycie danych i struktur w całym dokumencie. Schematy mogą posłużyć do kontroli poprawnej budowy dokumentu, lecz znacznie wychodzą poza zakres zagadnień strukturalnych, pozwalając programistom kontrolować typy i zawartość składników i samych struktur. Zarówno DTD, jak i schematy są cennymi narzędziami, zaś do jednego dokumentu możemy zastosować jedne i drugie rozwiązanie. Znajomość różnic pomiędzy nimi pomoże nam budować bezpieczniejsze dokumenty.

Weryfikacja struktury za pomocą DTD

DTD jest pierwszym etapem zapewniania jakości dokumentu XML. Kontrola poprawności poprzez DTD potwierdza, iż dokument jest poprawnie zbudowany — to znaczy, że struktura dokumentu odpowiada strukturze zdefiniowanej w DTD. Kontrolę poprawności może przeprowadzić zewnętrzny analizator składni lub paser zawarty w edytorze lub systemach programowania, np. XML Spy czy VS.NET.

DTD są starszą formą kontroli poprawności dokumentów XML obsługiwaną przez wszystkie parsery języka XML. Są analizowane składniowo i kontrolowane przed przetworzeniem schematów. Niestety, DTD posiadają też znacznie mniejsze możliwości niż schematy. Nadal są jednak przydatne do wymuszania jednolitej struktury dokumentów. Znacząca część tej wartości bierze się ze zdolności zewnętrznych DTD do wymuszania jednakowych struktur dokumentów XML tworzonych w różnych miejscach organizacji lub u partnerów biznesowych. Niestety, ta ważna zaleta niesie ze sobą znaczące zagrożenia bezpieczeństwa, dodatkowo zwiększone w schemacie XML.

W wielu przypadkach zewnętrzne DTD są przechowywane w serwerach poza zasięgiem kontroli lokalnych twórców dokumentów XML. W takich przypadkach wywołanie DTD oznacza korzystanie ze źródła, które niekoniecznie jest bezpieczne. Jeśli takie nie jest, wówczas przechwycona przez hakera definicja DTD może lawinowo rozprzestrzenić luki w zabezpieczeniach w innych organizacjach. Dotyczy to zwłaszcza przypadków, gdy typy `ENTITY` i `NOTATION` są powszechnie używane — chociażby przy tworzeniu baz danych multimedialnych. Na przykład, zmiana typu jednostki z `CDATA` na `ENTITY` pozwoli na użycie w niej danych znakowych bez generowania błędów. Jednakże zarazem umożliwi to przeniesienie aplikacji wiersza poleceń przez system XML do komputera docelowego bez kontroli poprawności zabezpieczeń ze strony systemu XML.

Kontrola spójności danych za pomocą schematu

Pokazaliśmy, że sam schemat niekoniecznie jest najlepszym narzędziem do kontroli poprawności jednolitości i przydatności danych wejściowych. Nie znaczy to, że schematy nie są przydatne. Nadal pozostają cennym narzędziem wymuszania zgodności dokumentów XML ze standardami i jednolitości dokumentów w różnych organizacjach.

Jedną z zalet schematów jest fakt, iż same w sobie są poprawnymi dokumentami XML. Gdy dokument zawierający schemat wywołuje analizator składni kontrolujący poprawność, wówczas schemat jest kontrolowany, zanim zostanie wykorzystany do kontroli

poprawności samego dokumentu. Poziom spójności osiągany przez taką kilkustopniową kontrolę poprawności jest znaczący.

Aktualny problem ze schematami polega na braku ostatecznego zatwierdzenia standardów. W chwili obecnej używane są trzy główne odmiany schematów. Schematami XML są:

- **XML Data Reduced (XDR)** — używany w pierwszej wersji protokołu SOAP (Simple Object Access Protocol) i obsługiwany przez Microsoft.
- **Regular Language description for XML (RELAX)** — uproszczony schemat, zaprojektowany do łatwego przechodzenia pomiędzy DTD i schematami. Dodatkowe informacje znajdziemy pod adresem *http://www.xml.gr.jp/relax*.
- **Schemat XML W3C** — ten schemat mają na myśli wszyscy mówiący o „standardowym schemacie". Na ostateczną wersję schemat XML organizacji W3C również niecierpliwie oczekuje wielu programistów. Dodatkowe informacje znajdziemy pod adresem *http://www.w3.org/TR/xmlschema-0*.

Wspólnym problemem DTD i schematów jest kwestia zewnętrznych odwołań. Podobnie jak DTD, wiele schematów to zewnętrzne zasoby, z którymi łączy się duża liczba różnych dokumentów XML. Ponieważ schematy kontrolują dane i struktury o wiele bardziej rygorystycznie niż DTD, potencjalne zagrożenie ze strony schematów przejętych przez hakerów jest większe. Z uwagi na to, na twórcach dokumentów spoczywa jeszcze większa odpowiedzialność za zapewnienie, by wszystkie używane schematy zewnętrzne pochodziły z bezpiecznych serwerów utrzymywanych przez zaufanych partnerów.

Metody i mechanizmy kontroli poprawności online

Gdy nadchodzi pora kontroli poprawności dokumentów XML, możemy postąpić na jeden z wielu sposobów. Wszystkie opcje łączy jeden wymóg: kontrolujący poprawność parser (analizator składni) musi odczytać wszystkie wiersze dokumentu i zwrócić raport o wszelkich naruszeniach poprawności lub o dobrze skonstruowanej strukturze. Wszystkie dostępne narzędzia do kontroli poprawności XML-a spełniają to najprostsze kryterium. Oprócz niego istnieje wiele opcji skorzystania z parsera sprawdzającego poprawność. Może to być autonomiczny produkt lub składnik pakietu do edycji lub programowania. Praktycznie wszystkie narzędzia kontrolujące poprawność są w stanie opierać kontrolę na DTD, co stanowi jedną z zalet tego starszego mechanizmu. Poniższe podpunkty przedstawią kilka najłatwiej dostępnych narzędzi online pozwalających na kontrolę poprawności w oparciu o DTD i schematy.

XML Spy 4.3

Program XML Spy 4.3, stworzony przez firmę Altova, jest środowiskiem programowania dla XML-a i obejmuje kontrolę poprawności na podstawie DTD i schematów, dostęp do baz danych ODBC (Open Database Connectivity) oraz całkowicie zintegrowane środowisko programowania. Darmową, 30-dniowa wersję próbną możemy znaleźć pod adresem *http://www.xmlspy.com*.

W3C: Validator for XML Schema

Udostępniony przez W3C program kontroli poprawności dokumentów XML na podstawie schematu możemy znaleźć pod adresem *http://www.w3.org/2001/03/webdata/xsv*. Serwis ten zapewnia możliwość korzystania ze schematów dostępnych poprzez WWW, jak również znajdujących się za zaporą firewall przedsiębiorstwa (patrz: rysunek 4.2).

Rysunek 4.2. *Ekran powitalny programu W3C Validator for XML Schema*

Programiści mogą przesyłać do niego dokumenty XML na dwa sposoby: podając ogólnodostępny adres URI lub ładując plik z sieci lokalnej. Poprawnie zbudowany dokument zwróci jedynie prosty ekran z komunikatem, jak widać na rysunku 4.3.

Rysunek 4.3. *Notatka zwrócona dla poprawnie zbudowanego dokumentu*

Gdy program sprawdzający poprawność przeanalizuje dokument z błędami, otrzymamy ekran z komunikatami o błędach, podający typ błędu i jego położenie w pliku, jak pokazano na rysunku 4.4.

Rysunek 4.4. *Dokument zawierający błędy powoduje wygenerowanie komunikatu o błędzie*

Powinniśmy zwrócić uwagę na jedną ważną właściwość programu kontrolującego poprawność — podobnie jak wiele staroświeckich narzędzi tego typu, zatrzymuje się po natrafieniu na pierwszy błąd. Jeśli nasz dokument jest długi i zawiera ich sporo, wówczas zapewnienie poprawnej budowy dokumentu będzie wymagało wielokrotnego jego uruchomienia.

Formularz kontroli poprawności XML-a z Brown University

Programiści z Brown University stworzyli formularz pozwalający kontrolować poprawność XML-a. Formularz ten jest dostępny online pod adresem *http://www.stg.brown.edu/service/xmlvalid/*. Krótkie dokumenty możemy skopiować i wkleić bezpośrednio na stronie; większe dokumenty są wywoływane poprzez adres.

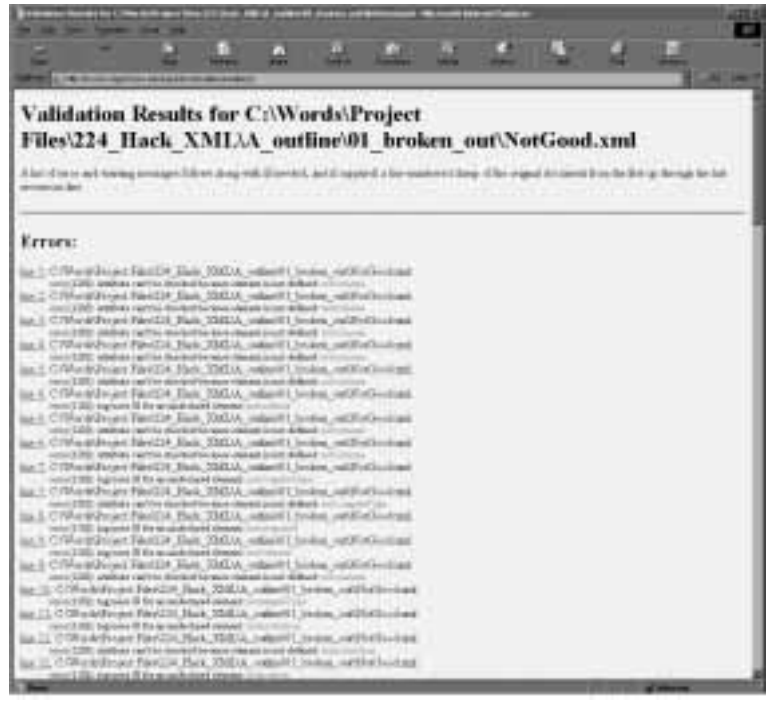
Gdy po raz pierwszy otworzymy formularz kontroli poprawności z Brown University, zobaczymy prosty ekran interfejsu jak przedstawiono na rysunku 4.5.

Od razu widać, że grupa programistów z Brown podeszła do kontroli poprawności zupełnie inaczej niż w przypadku innych takich serwisów online, ponieważ dokument zaakceptowany przez programy W3C i RUWF zwraca w przypadku Brown University wyniki jak te z rysunku 4.6. Jak widzimy, dokument został surowo oceniony.

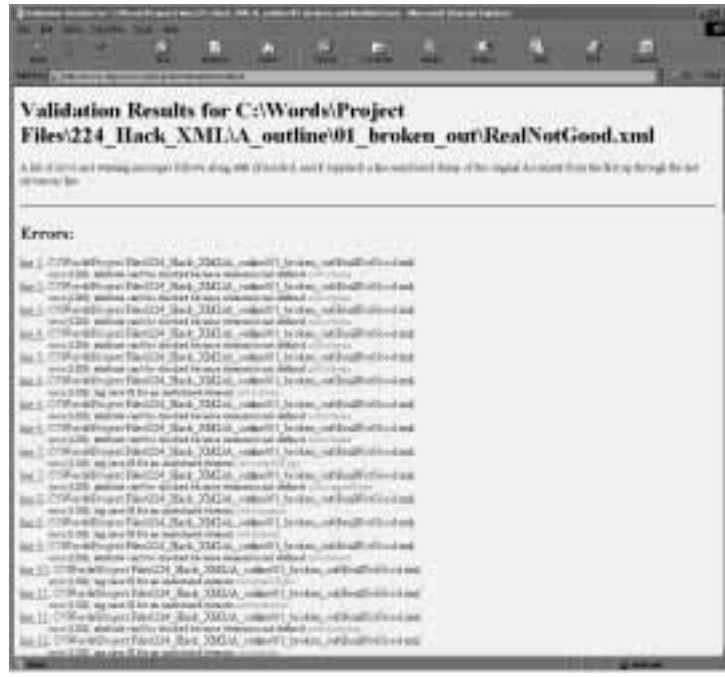
Większość błędów wyszczególnionych przez ten formularz dla innych narzędzi byłoby co najwyżej ostrzeżeniami. Jeśli potrzebujemy absolutnej pewności, że dokument będzie działać, wówczas narzędzie z Brown University pozwoli nam tę pewność uzyskać, lecz inne narzędzia powiedzą, czy dokument będzie działać czy nie przy wykorzystaniu znacznie mniejszej liczby wierszy komentarza. Jednym z problemów, jakie napotyka się w przypadku takiego podejścia, jakie zastosowano w Brown University, jest kontrola plików zawierających znane problemy. Plik, który spowodował wygenerowanie błędów w narzędziach W3C i RUWF, w narzędziu z Brown zwrócił ekran przestawiony na rysunku 4.7. Proszę zwrócić uwagę, że wyniki dla pliku poprawnego i niepoprawnego są do siebie podobne.

Raporty generowane przez dwa wprowadzone pliki niewiele się od siebie różnią. W wynikach dla „złego" pliku błędy znalezione przez inne narzędzia kontroli poprawności

Rysunek 4.5. *Formularz kontroli poprawności online z Brown University*



Rysunek 4.6. *Plik zaakceptowany przez inne programy kontrolujące poprawność został mocno skrytykowany w Brown*



Rysunek 4.7. *Wyniki kontroli poprawności złego pliku w narzędziu Brown University*

zagubiły się w liście ostrzeżeń. Formularz Brown University jest narzędziem dokładnym, lecz powinniśmy go używać w połączeniu z innymi narzędziami, a nie zamiast nich.

Nie wspierane narzędzie Microsoftu do kontroli poprawności XML-a

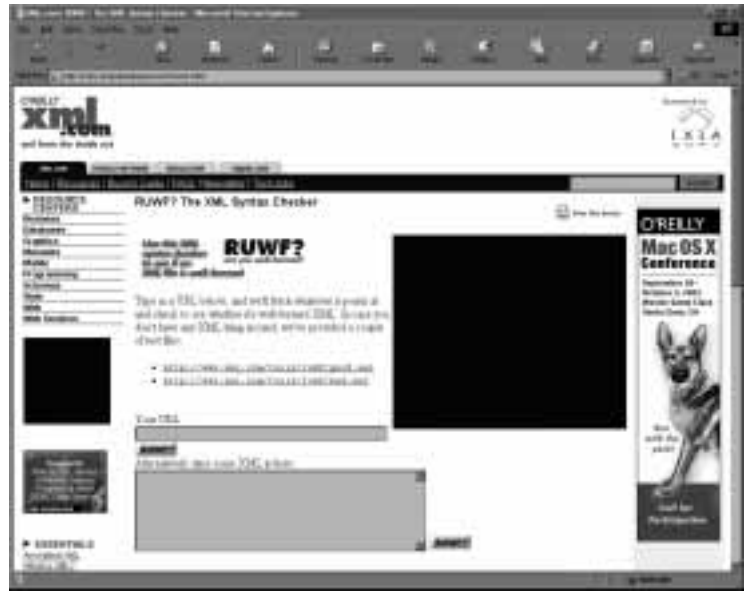
Microsoft oferuje narzędzie sprawdzające poprawność dokumentów XML na podstawie zarówno DTD, jak i schematów. Możemy je znaleźć pod adresem *http://msdn.microsoft.com/downloads/sample.asp?url=/msdn-files/027/000/537/msdncompositedoc.xml*.

Narzędzie kontroli poprawności XML-a XML.com

Witryna WWW XML.com oferuje narzędzie online do kontroli poprawności XML-a pod adresem *http://xml.com/pub/a/tools/ruwf/check.html*. Programiści mogą albo wprowadzić URL, który zostanie sprawdzony, albo skopiować i wkleić kod XML do okna na stronie (patrz: rysunek 4.8). Ekran powitalny daje dostęp do narzędzia kontrolującego poprawność i odnośniki do innych narzędzi.

Poprawnie zbudowany dokument wklejony do okna XML na tej stronie, zwraca małe okienko z gratulacjami, co pokazano na rysunku 4.9.

Jeśli dokument nie jest dobrze zbudowany, wówczas RUWF wyświetla okno zawierające listę wszystkich błędów wraz z ich umiejscowieniem. Komunikaty o błędach RUWF nie są tak dokładne jak w przypadku Brown University, lecz dokument przetwarzany jest od początku do końca, nawet po znalezieniu błędu — takie podejście umieszcza to narzędzie pośrodku drogi pomiędzy dwoma poprzednimi opisanymi tu narzędziami online.



Rysunek 4.8. *Kontroler składni RUWF pozwala na wprowadzanie adresu URL lub wklejenie tekstu, lecz nie umożliwia ładowania plików*

Rysunek 4.9. *Kontroler składni RUWF dla dokumentów poprawnie zbudowanych wyświetla tylko prosty komunikat*

Wieloschematowe narzędzie kontroli poprawności XML-a firmy Sun

Oprogramowanie firmy Sun, oparte na Javie, dokonuje kontroli poprawności względem schematów RELAX NG, Relax Namespace, RelaxCore, TREX, definicji DTD i podzbioru schematu XML Schema Part 1. Jest ono podstawą oprogramowania kontrolującego poprawność w Apache Web Suite. Znajdziemy je pod adresem *http://www.sun.com/software/xml/developers/multischema/*.

Podsumowanie

Kontrola poprawności XML-a jest wieloetapowym procesem, który zapewnia poprawność strukturalną i spójność logiczną dokumentu oraz zgodność danych z wymogami aplikacji. Żaden z tych etapów nie jest wymagany dla dokumentu XML i danych z nim skojarzonych, lecz wszystkie razem chronią aplikację i system przed rozmyślnym atakiem i nieumyślnymi błędami.

Standardy, które definiują XML, obejmują dwie odrębne, aczkolwiek podobne metody kontroli poprawności: definicje typu dokumentu (DTD) i schematy. DTD zajmują się strukturą dokumentu XML (czy jest poprawnie zbudowany) i mają tylko podstawowe możliwości kontroli typu danych zdefiniowanych dla dokumentu. DTD są mechanizmem starszym i bardziej rozpowszechnionym niż schematy, przez co obsługują je niemal wszystkie analizatory składni, przeglądarki i narzędzia programistyczne dla XML-a. Definiując strukturę danych używanych w dokumencie XML za pomocą DTD, możemy ujednolicić dokumenty tworzone w różnych miejscach organizacji lub w różnych organizacjach. Weryfikacja poprawnej budowy dokumentu XML za pomocą jednej lub kilku definicji DTD zapewnia wysoki poziom pewności, iż struktura danych użyta w dokumencie nie zawiera żadnych niekonsekwencji.

Oprócz zalet, DTD posiadają trzy poważne wady. Po pierwsze, nie są zapisane w standardowej gramatyce języka XML ani żadnego innego języka. Oznacza to, że narzędzia kontrolujące poprawność poprzez DTD nie mogą kontrolować samych DTD. Po drugie, DTD umożliwiają jedynie najbardziej podstawową kontrolę definicji danych, które zapełnią struktury zdefiniowane w dokumencie XML. DTD nie posiadają żadnej funkcjonalności pozwalającej na definiowanie rozmiarów lub zawartości danych. Ponadto te same zewnętrzne DTD, które zapewniają spójność strukturalną dokumentów w wielu organizacjach, mogą stanowić poważne zagrożenie zabezpieczeń, jeśli serwer, na którym są przechowywane, zostanie w jakikolwiek sposób infiltrowany.

Schematy zostały zaproponowane jako mechanizm mający zastąpić DTD w procesie kontroli poprawności dokumentów. Schematy umożliwiają taką samą kontrolę struktur jak DTD, lecz posiadają wiele dodatkowych możliwości kontroli zawartości danych w dokumentach XML. Schematy mogą, na przykład, służyć do ograniczania typów (liczbowe, łańcuchowe itp.) i długości danych. Schematy mogą istnieć w tym samym dokumencie co DTD i często są stosowane do ograniczania typów danych w strukturach definiowanych przez DTD.

Schematy są zapisywane w standardowej gramatyce XML-a, co umożliwia kontrolę ich poprawności za pomocą tych samych analizatorów składniowych, co używane wobec zwykłych dokumentów XML. Podobnie jak DTD, schematy mogą mieścić się wewnątrz dokumentu XML w wyznaczonym miejscu na początku kodu, lub na zewnątrz, adresowane przez nazwę pliku i jego położenie. Zewnętrzne schematy umożliwiają czerpanie korzyści ze scentralizowanych schematów definiujących struktury danych i treść w różnych jednostkach organizacyjnych. Jednakże, podobnie jak w przypadku DTD, ta możliwość niesie ze sobą jedną z dwóch poważnych wad.

Pierwszą z nich jest fakt, iż schematy nie zostały jak dotąd w pełni znormalizowane. Wprawdzie drastyczne zmiany w strukturach lub funkcjach schematów są bardzo mało prawdopodobne, lecz zmiany wystarczająco duże, by „złamać" istniejące schematy, są możliwe. Brak formalnego standardu powoduje, iż nadal istnieją narzędzia do tworzenia XML-a i analizatory składni, które nie weryfikują schematów — ich autorzy czekają na „twardy" standard, zanim przeprowadzą weryfikację kodu. Druga poważna wada schematów jest taka sama jak w przypadku DTD — schematy zewnętrzne, wyjątkowo przydane przy ujednolicaniu struktur danych i treści w różnych wydziałach lub nawet przedsiębiorstwach, wprowadzają ryzyko pojawienia się nieautoryzowanego kodu w przypadku infiltracji źle chronionego serwera, z którego są pobierane.

Gdy dla dokumentu XML zostanie już zweryfikowana poprawna budowa i właściwe definicje danych, wówczas wewnętrzne mechanizmy kontroli poprawności XML-a kończą swoje zadanie. Niestety, zanim uznamy aplikację i dane oparte na dokumencie XML za zabezpieczone przed nieumyślnymi szkodami lub zamierzonym atakiem, niezbędnych jest jeszcze kilka kroków. Twórca aplikacji musi zbudować procedury kontroli poprawności danych wprowadzanych do aplikacji (nieważne, czy pochodzą z klawiatury, z transferu plików czy też ze strumienia danych wygenerowanego przez komputer). Kontrolę poprawności danych wejściowych możemy podzielić na trzy podstawowe etapy: sprowadzenie strumienia danych wejściowych do postaci kanonicznej, kontrolę poprawności kodu Unicode i kontrolę poprawności dokumentów i komunikatów.

Dwa pierwsze kroki są niezbędne, ponieważ znaki mogą być reprezentowane na wiele różnych sposobów i poddawane translacji podczas przechodzenia z ekranu i klawiatury przez transmisję sieciową i przetwarzanie wewnątrz aplikacji. Te różne metody reprezentacji umożliwiają hakerom atak tekstem jawnym, polegający na przesyłanie łańcuchów znakowych sterujących aplikacją lub systemem w sposób niezgodny z intencjami twórcy aplikacji. Większość aplikacji i systemów zawiera procedury szukające łańcuchów znaków, które mogłyby dać dostęp do funkcji niedostępnych dla zwykłych użytkowników, lecz ataki tekstem jawnym zakładają, że te procedury dopasowywania łańcuchów polegają na jednym, określonym sposobie reprezentacji znaków, podczas gdy system i aplikacja dopuszczają użycie kilku różnych opcji kodowania znaków. Stosując mniej popularną metodę kodowania, hakerzy usiłują „ukryć na widoku publicznym" szkodliwy ładunek rozkazów.

Najczęściej stosowaną w języku angielskim reprezentacją znaków jest ASCII — 8-bitowy zestaw znaków używany przez większość komputerów osobistych i wiele serwerów internetowych. Jednakże do zapisu w bazie danych i przetwarzania w aplikacjach wielu programistów i wiele języków programowania dokonuje translacji znaków ASCII na jeden z zestawów znaków Unicode — zbioru międzynarodowych standardów dla danych alfanumerycznych. Ponieważ wiele języków pisanych używa większej liczby znaków niż język angielski oraz wiele spośród nich zawiera znaki bardziej złożone niż w alfabecie angielskim, Unicode wykorzystuje znaki kodowane przy wykorzystaniu 16 lub 32 bitów, w zależności od języka.

Problemy pojawiają się, gdy określone elementy, wspólne dla wszystkich języków pisanych (między innymi spacja, powrót karetki, koniec wiersza i znaki sterujące wskazujące początek i koniec transmisji) posiadają reprezentacje w zestawach znaków o różnych długościach. Od programisty zależy zapewnienie, by znaki ASCII były przekładane na najkrótsze możliwe reprezentacje Unicode, aby uniknąć zamieszania i pojawienia się niezamierzonych rozkazów.

Po dokonaniu translacji znaków do Unicode musimy stworzyć procedury zapewniające, by ich reprezentacje były spójne i nie zawierały znaków i elementów sterujących, które mogłyby spowodować zapisanie przez aplikację lub system bezsensownych danych lub naruszenie bezpieczeństwa w określony sposób. Ten krok jest niezbędny, ponieważ większość języków programowania pozwala na transmisję znaków Unicode w postaci liczbowej lub nazwy oprócz reprezentacji ASCII. Wprawdzie liczbowe reprezentacje mogą obejść proste procedury zabezpieczeń szukające określonych zakazanych łańcuchów znakowych, lecz kontrola poprawności po translacji całego łańcucha

do Unicode zapewnia identyfikację i neutralizację zakazanych instrukcji i sekwencji niezależnie od początkowego odwzorowania transmisji.

Na koniec, po weryfikacji i kontroli poprawności XML-a oraz po poprawnej translacji łańcucha wejściowego do niegroźnego Unicode, wejściowy łańcuch znaków może zostać zweryfikowany jako poprawny logicznie dokument lub komunikat mający sens dla aplikacji. Często oznacza to kontrolę łańcucha, aby, na przykład, upewnić się, czy wprowadzony numer karty kredytowej zgadza się z wzorcem liczbowym dla danego typu karty. Ten etap weryfikacji jest wyjątkowo ważny, lecz musi zostać przeprowadzony bardzo ostrożnie, z uwagi na jego naturę, wymagającą intensywnego wykorzystania mocy obliczeniowej komputera. Wielu programistów może uważać, iż schemat XML oferuje prostą, wiarygodną metodę tworzenia mechanizmów weryfikacji komunikatów, lecz natura XML-a (wymagająca wywołanie schematu z pamięci i parsowanie za każdym razem, gdy weryfikacja jest niezbędna) powoduje, iż opcja ta nie jest pożądana dla aplikacji wymagających weryfikacji tysięcy komunikatów w ciągu minuty.

Niezależnie od wybranej metody implementacji weryfikacja i kontrola poprawności na wszystkich etapach, od poprawnie zbudowanej struktury aż do kontroli poprawności metody, dają aplikacje i dane wejściowe, które spełniają wymogi organizacji, a zarazem minimalizują zagrożenie atakami hakerów.

Rozwiązania w skrócie

Definicje typu dokumentu i poprawnie zbudowane dokumenty XML

- Definicje typu dokumentu (DTD) służą do weryfikacji, czy dokument XML jest poprawnie zbudowany (poprawny strukturalnie).
- DTD nie są wymagane w żadnych dokumentach XML.
- DTD mogą być częścią dokumentu XML lub odrębnymi dokumentami wywoływanymi przez adres URI (*Uniform Resource Indicator*) w dokumencie XML.
- DTD nie są pisane z wykorzystaniem standardowej gramatyki XML-a.
- DTD nie nakładają ograniczeń na zawartości elementów XML-a, a jedynie definiują strukturę dokumentu.
- DTD mogą być używane w dokumencie XML razem ze schematem.

Schemat i poprawne dokumenty XML

- Schematy XML są stosowane do wymuszenia struktury danych opisanych w dokumencie XML. Schemat może też nakładać ograniczenia na dane zawarte w poszczególnych elementach.
- Schematy nie są wymagane w żadnych dokumentach XML.

- Schemat może być częścią dokumentu XML lub odrębnym elementem wywoływanym przez odnośnik do URI w dokumencie XML.
- Schemat może być używany w dokumencie XML razem z DTD.

Wprowadzenie do ataków tekstem jawnym

- Ataki tekstem jawnym wykorzystują istnienie różnych metod reprezentacji znaków, wspólnych dla różnych języków i systemów.
- Ataki tekstem jawnym często wykorzystują reprezentacje szesnastkowe popularnych znaków sterujących lub systemowych (np. łańcucha / . . / . . /), brane z rzadko spotykanych 32-bitowych reprezentacji znaków Unicode, aby uniknąć wykrycia i neutralizacji przez procedury zabezpieczeń stosujące dopasowywanie do wzorców.
- Możemy zabezpieczyć się przed atakami tekstem jawnym przez podwójny proces sprowadzania do postaci kanonicznej (który zapewni translację wszystkich nadchodzących łańcuchów znakowych do najprostszej możliwej reprezentacji Unicode) i przez weryfikację Unicode.

Sprawdzanie poprawności XML-a

- Jeśli stosowany jest analizator składniowy sprawdzający poprawność DTD, wówczas DTD są przetwarzane przed schematem, co zapewnia poprawną budowę (strukturę) dokumentu.
- Poprawność dokumentów XML jest sprawdzana na podstawie schematu po kontroli na podstawie DTD. Schematy wymuszają jednolitość danych i treści dla struktur danych zdefiniowanych przez dokument XML.
- Programiści aplikacji odpowiedzialni są za sprowadzanie do postaci kanonicznej zapewniającej translację wszystkich nadchodzących łańcuchów znakowych z ASCII do najkrótszej możliwej reprezentacji Unicode.
- Łańcuch kanoniczny Unicode po utworzeniu musi zostać zweryfikowany jako niegroźny — nie niosący łańcuchów próbujących w niepowołany sposób uruchomić aplikację lub uzyskać dostęp do nieautoryzowanych plików.
- Ostatnim krokiem w procesie kontroli poprawności jest sprawdzenie poprawności dokumentu lub komunikatu, kiedy to kontrolowana jest przydatność nadesłanego łańcucha dla elementu danych, który jest jego celem. Na tym etapie musimy zwrócić uwagę, by metoda kontroli poprawności była wystarczająco wydajna, aby użytkownicy nie mieli problemów z opóźnieniami odpowiedzi systemu.

Pytania i odpowiedzi

Poniższe pytania, na które odpowiadają autorzy tej książki, mają na celu sprawdzenie poziomu zrozumienia przez Czytelnika zagadnień przedstawionych w tym rozdziale oraz pomoc w praktycznym zastosowaniu tych zagadnień.

P: Czy zawsze muszę definiować schemat dla mojego dokumentu XML?

O: Nie, schemat nie zawsze jest potrzebny. Schematy są bardzo przydatne, gdy musimy skontrolować poprawność dokumentu, zazwyczaj przy wymianie dokumentów XML przez Internet. Dokonywanie kontroli poprawności za każdym razem może wydać się doskonałym pomysłem, lecz jest kosztowną operacją, która może zadławić serwer WWW. Podczas publikowania dokumentów XML w sieci schemat zwykle nie jest potrzebny, lecz stanowi doskonałe narzędzie do dokumentowania XML-a.

P: Czy zawsze muszę definiować DTD dla mojego dokumentu XML?

O: Nie. Podobnie jak schematy, DTD są całkowicie opcjonalne dla dowolnego dokumentu XML. DTD wymagają nakładów mocy obliczeniowej komputera podobnych jak dla schematów, lecz ponieważ DTD nie kontrolują zawartości składników i struktur, więc mniej zachęcają do kontroli poprawności dokumentu przy każdym jego wywołaniu.

P: Którego zestawu znaków Unicode mam używać?

O: To zależy od języka obsługiwanego przez aplikację. Jeśli jest to język angielski, wówczas Unicode UTF-8 zawiera najkrótsze reprezentacje wszystkich potrzebnych znaków. Jeśli aplikacja musi obsługiwać też inne języki, wówczas możemy sprawdzić na stronie WWW Unicode (*http://www.unicode.org*), jaki zestaw znaków jest poprawny, lecz nadal warto upewnić się, czy wszelkie znaki niedrukowalne są odwzorowywane na najkrótsze możliwe reprezentacje.

P: Po co mam pisać własne procedury kontroli poprawności łańcuchów wejściowych? Czy dobry schemat nie poradzi sobie z tym za mnie?

O: Głównym powodem do napisania własnych procedur jest wydajność. Schematy muszą być wywoływane i analizowane składniowo za każdym razem, gdy dokument jest kontrolowany. Na popularnej stronie WWW zawierającej katalog mogą to być setki lub tysiące powtórzeń na minutę. Moc obliczeniowa zużywana na wywoływanie, parsowanie i kontrolę poprawności na podstawie schematu za każdym razem, gdy klient będzie chciał złożyć zamówienie, jest w takim wypadku nie do zaakceptowania.

P: Czy są jakieś sposoby na śledzenie najnowszych typów ataków używanych przez hakerów?

O: Strona domowa CERT (*http://www.cert.org*) jest najlepszym źródłem informacji o wszelkich typach ataków komputerowych i sposobach obrony przed nimi. Ten serwis oferuje powiadamianie pocztą elektroniczną o nowych atakach i bazę danych pozwalającą uaktualnić naszą wiedzę o istniejących lukach w zabezpieczeniach.