Emulação de Ataques do Tipo *XPath Injection* para Testes de *Web Services* usando Injeção de Falhas

Marcelo I.P. Salas[1], Eliane Martins[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 676 – 13083-970 – Campinas – SP – Brazil

{marcelopalma@ic.unicamp.br,eliane}@ic.unicamp.br

Abstract. *This paper describes the use of a fault injector to emulate XPath Injection attacks for security testing of Web Services. XPath Injection is a kind of injection attacks, one of the most exploited by attackers, and its consequences are harmful when well succeeded. One of the standards to ensure security in the context of Web Services is the WS-Security (WSS), which, among other mechanisms, uses Security Tokens for access control to messages exchanged between services. The results showed that the use of this mechanism improves the detection of XPath injection, but is still not enough to ensure 100% protection against this type of attack.*

Resumo. *Este artigo descreve o uso de um injetor de falhas para emular ataques de tipo XPath Injection para testar a segurança de Web Services. XPath Injection é um dos ataques de injeção, que são dos mais explorados, além de serem considerados como um dos mais perigosos, quando são bem sucedidos. Um dos padrões para garantir a segurança no contexto de Web Services é o WS-Security (WSS), o qual, entre outros mecanismos, utiliza credenciais de segurança (Security Tokens) para garantir o controle de acesso às mensagens trocadas entre serviços. Os resultados mostraram que o uso desse mecanismo melhora a detecção de XPath Injection, mas ainda não é suficiente para garantir 100% de proteção contra esse tipo de ataque.*

1. Introdução

Os *Web Services* (WS) permitem a interoperabilidade entre sistemas de software desenvolvidos em linguagens de programação diferentes e executados sobre qualquer plataforma [Moorsel et al. 2009]. Utilizando *Web Services*, uma aplicação (cliente) pode invocar outra (servidora), independentemente do local onde resida a aplicação servidora, ou da plataforma em que esta execute. Um WS pode oferecer diversas operações a seus clientes, as quais são descritas usando WSDL (*Web Service Description Language*). Os WS interagem através da troca de mensagens seguindo o protocolo SOAP[1]. Tanto o WSDL quanto as mensagens SOAP utilizam o formato **XML** (*Extensible Markup Language*). O transporte de dados é realizado normalmente via protocolo HTTP (*Hypertext Transfer Protocol*) ou via HTTPS (*Hypertext Transfer Protocol Secure*), no caso de conexões seguras [Hartman et al. 2003].

A realização de negócios e serviços através da Internet faz com que a segurança (*security*) se torne um aspecto ainda mais crítico, de vez que informações importantes

[1] Inicialmente SOAP era abreviatura de *Simple Object Access Protocol*, mas a W3C abandonou tal definição, e hoje em dia SOAP é simplesmente o nome do protocolo, e não mais um acrônimo.

para as empresas circulam através da Internet. Pesquisas realizadas pelo Ponemon Institute e Symantec[2] analisaram os custos devidos a violações de dados de 51 organizações nos EUA em 2010, e revelaram que ataques maliciosos e criminosos se tornaram mais freqüentes, sendo responsáveis por 31% da ocorrência de violação de dados. Além do aumento do número de ataques, o estudo chama a atenção para o alto custo dos mesmos: a diferença entre o custo de violações maliciosas e não maliciosas cresceu mais de dez vezes, ou seja, suas conseqüências se tornam cada vez mais severas.

No caso de *Web Services*, as características que os tornam atrativos, tais como maior acesso a dados e conexões dinâmicas entre aplicações, apresentam novos desafios para a segurança. Além das ameaças tradicionais associadas aos protocolos de rede, tem-se que contar ainda com novas ameaças associadas a novos protocolos e serviços, como SOAP e XML. Um exemplo são os chamados ataques de injeção (*injection flaws*), dos mais explorados em 2010, segundo o relatório anual de segurança do Open Web Application Security Project (OWASP)[3]. Esses ataques também foram considerados como dos mais perigosos, de uma lista de 25 ataques publicados em 2011 por CWE e SANS[4]. Ataques de injeção ocorrem quando dados fornecidos pelo usuário são enviados a um interpretador como parte de um comando ou consulta. Os dados hostis do atacante enganam o interpretador para executar comandos mal intencionados ou manipular dados. A proteção contra ataques desse tipo requer mecanismos de segurança aplicados às mensagens SOAP, para garantir seu transporte até o destinatário final. Foram, para tanto, definidos uma série de padrões para segurança de mensagens, dentre os quais o WS-**Security** (**WSS**) [Holgersson e Soderstrom 2005]. Com base no uso de assinaturas digitais (*XML Signature*), credenciais de segurança (*Security Tokens*) e criptografia de documentos XML (*XML Encription*), WSS visa garantir a segurança na troca de mensagens SOAP.

Uma forma muito comum de determinar se aplicações são vulneráveis a ataques é através do uso de *vulnerability scanners* (VS). Essas ferramentas usam testes de penetração para introduzir ataques e com isso detectar vulnerabilidades. Essas ferramentas diferem em termos do tipo de ataques que elas são capazes de emular. Um estudo recente utilizou várias VS, comerciais e de código aberto, com o objetivo de detectar falhas de segurança em *Web Services* [Vieira et al.2009]; com esse estudo os autores puderam constatar que, para os VS utilizados, houve baixa cobertura das vulnerabilidades conhecidas, e um alto número de falsos positivos. Outra constatação desse estudo é que muitos dos WS em uso são pouco testados em termos de segurança (foram avaliados 300 serviços).

Nossa abordagem usa injetores de falhas para testar a segurança na troca de mensagens entre *Web Services* e seus clientes, ao invés de usar *vulnerability scanners*. O uso de injetores visa obter maior cobertura de ataques, pois permite emular diversos tipos de ataques, por um lado, e também permite variar os parâmetros e dados injetados para emular um determinado ataque. Para reduzir o número de falsos positivos, utilizamos um conjunto de regras, como proposto em [Antunes e Vieira 2009], com base em resultados obtidos de várias fontes.

---

[2] http://www.slideshare.net/symantec/2010-annual-study-us-cost-of-a-data-breach
[3] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[4] http://cwe.mitre.org/top25/index.html#Listing

Para mostrar a utilidade da abordagem, foram realizados testes de 10 *Web Services* reais, sendo 5 deles utilizando o padrão WSS e os outros 5, não. Os 10 serviços foram testados em presença de *XPath Injection*, um tipo de ataque de injeção que explora o aplicativo XPath (XML Path Language), ele permite a consulta ou navegação em documentos XML. Os resultados mostraram que o percentual de ataques bem sucedidos passou de 100% a 28% com o uso do WSS.

O texto está organizado da seguinte forma: a Seção 2 apresenta os conceitos e a tecnologia *Web Services*, além dos desafios da Segurança nesse contexto; Seção 3 apresenta a abordagem utilizada. A Seção 4 descreve o estudo experimental realizado e os resultados obtidos estão na Seção 5. A Seção 6 conclui o trabalho, mostrando suas principais contribuições e apontando direções para trabalhos futuros.

2. Aspectos de Segurança em *Web Services*

A segurança é uma qualidade de sistema que garante a ausência de acesso ou manipulação, não autorizados, ao estado do sistema [Avizienis et al. 2004]. A segurança tem como principais atributos a **confidencialidade** (a informação só deve ser revelada para usuários autorizados), a **disponibilidade** (o acesso ao sistema não pode ser negado, de forma maliciosa, a usuários autorizados), e a **integridade** (a informação não pode ser modificada por usuários não autorizados) [Holgersson e Soderstrom 2005].

As violações de segurança dos sistemas ocorrem devido à exploração de vulnerabilidades existentes. **Vulnerabilidades** são falhas (*faults*) introduzidas, intencionalmente ou acidentalmente, durante o desenvolvimento do sistema. Existem inúmeras causas para a existência de vulnerabilidades, dentre as quais podemos citar a complexidade dos sistemas, bem como a falta de mecanismo para verificação das entradas fornecidas. Um **ataque** explora as vulnerabilidades do sistema, de forma maliciosa ou não, podendo comprometer as propriedades de segurança do sistema. O resultado de um ataque bem sucedido é uma **intrusão** no sistema. A Figura 1 ilustra esses conceitos.

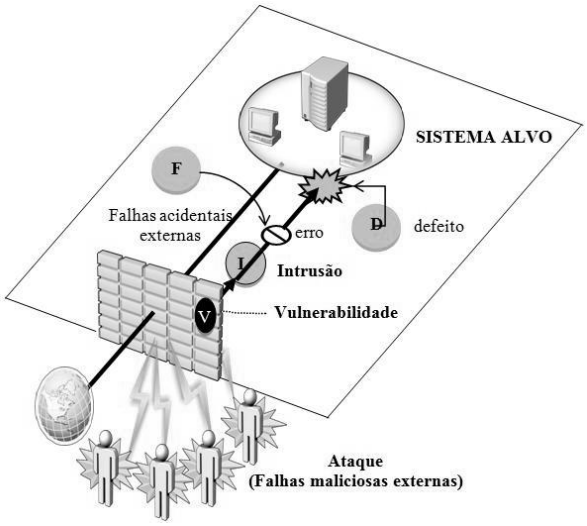

Figura 1. Ameaças à segurança [Cachin e Camenisch 2000].

2.1 Alguns aspectos de segurança em *Web Services*

A segurança em *Web Services* pode ser tratada *ponto-a-ponto* ou *fim-a-fim* [IBM e Microsoft 2011], sendo que diferentes padrões foram propostos para cada contexto. No contexto *ponto-a-ponto* procura-se garantir a segurança no transporte de dados; para isso, existem diversos padrões, dentre os quais o HTTPS, já citado, uma extensão do HTTP. A segurança *ponto-a-ponto* permite garantir a confidencialidade dos dados transportados, mas, no caso em que mensagens passem por WS intermediários antes de atingir o destinatário final, a segurança dessas mensagens não está garantida. A segurança *fim-a-fim* visa proteger a troca de mensagens SOAP entre clientes e servidores. Dentre os padrões propostos podemos citar XML-Signature [Eastlake et al. 2008], que define regras para gerar e validar assinaturas digitais[5] expressas em XML. XML-Encryption [Eastlake et al. 2002], que especifica o processo de encriptação dos dados e sua representação em XML. Ou ainda, Security Token [Lawrence et al. 2006-A], que comprova a identidade do cliente, para que este possa ter acesso aos serviços do servidor, usando credenciais de segurança. Tem-se também a especificação padronizada pelo OASIS, WS-Security (WSS) [Lawrence et al 2006-B], que define um conjunto de extensões ao protocolo SOAP, e utiliza os padrões XML-Signature, XML-Encryption e Security Token, oferecendo: (i) integridade, com o uso de assinaturas digitais para o total ou parte das mensagens, (ii) confidencialidade, permitindo que mensagens SOAP sejam cifradas no todo ou em parte, e (iii) autenticidade, com o uso de credenciais de segurança nas mensagens SOAP.

Além da segurança de mensagens, existem outras dimensões para segurança de aplicações baseadas em *Web Services*, tais como proteção de recursos e políticas de segurança; foram propostos padrões para esses diversos aspectos. Dado que nosso interesse está no WSS, não trataremos desses assuntos aqui. O leitor interessado pode consultar, por exemplo, [Mello et al. 2006] ou [Singhal et al. 2007] para uma introdução mais completa sobre o assunto.

2.2 Técnicas para detectar vulnerabilidades

Para desenvolver *Web services* seguros o provedor tem a sua disposição uma série de ferramentas, linguagens e técnicas. Deve também seguir as boas práticas de segurança, e escolher os padrões de segurança mais adequados. No entanto, é preciso também determinar se o *Web Service* implementado apresenta o nível de segurança desejado. Dado que a segurança pode ser comprometida devido à existência de vulnerabilidades, inúmeras técnicas de detecção de vulnerabiliddes estão disponíveis, devendo ser usadas tanto por provedores quanto por usuários de serviços.

Técnicas estáticas não necessitam execução do serviço ou aplicação em teste; as técnicas mais comuns são a análise estática automatizada e inspeção de código. Já as técnicas dinâmicas necessitam da execução da implementação. Nessa categoria, os Testes de Penetração, Fuzz Testing e Injeção de Falhas (IF) são as mais utilizadas.

Os Testes de Penetração simulam ataques com o intuito de revelar vulnerabilidades. Os Testes de Penetração automatizados são possíveis com o uso de ferramentas denominadas *Vulnerability Scanners* (VS). Existem diversos VS, tanto comerciais (por exemplo, HP Web Inspect, IBM Rational AppScan) quanto de código

[5] As assinaturas digitais visam garantir a integridade e confidencialidade de mensagens [Eastlake 2008].

aberto (e.g. WSDigger e WebScarab). As vulnerabilidades detectadas variam de uma ferramenta para outra. Uma avaliação de várias versões de VS comerciais mostrou que essas ferramentas têm como principais limitações a baixa cobertura das vulnerabilidades existentes e a alta porcentagem de falsos positivos [Antunes e Vieira 2009]. Dado que executamos a IF durante a execução, também utilizamos testes de penetração para detectar vulnerabilidades. À diferença dos testes usando VS, a nossa abordagem permite uma cobertura maior dos ataques possíveis.

Fuzz testing [Miller et al. 1995] é uma técnica que consiste em fornecer entradas inválidas, inesperadas ou aleatórias a um sistema e observar possíveis defeitos como colapso (crash) do cliente ou do servidor, ou lançamento de exceções imprevistas. É uma técnica muito usada para testar a segurança de sistemas computacionais. Como exemplos de trabalhos usando essa técnica para testar a segurança no contexto de *Web Services* podemos citar H-Fuzzing [Zhao et. al. 2009] e a ferramenta SQL Fuzzing [Garcia 2009]. Uma vantagem dos fuzz tester é que são úteis para revelar a presença de falhas mais difíceis de serem reveladas com testes criados manualmente, e que podem ser explorados por um atacante. No entanto, a cobertura de vulnerabilidades conhecidas pode ser baixa. Nossa abordagem permite maior controlabilidade dos ataques gerados.

A injeção de falhas consiste em introduzir, seja por hardware ou por software, falhas ou erros em um sistema e observar o seu comportamento [Arlat et al. 1990]. Existem diversas formas de injetar falhas em um sistema. A mais atrativa, do ponto de vista de custo de realização e facilidade de adaptação a diferentes sistemas e plataformas é a IF por software. Nesse caso, as falhas são introduzidas por um injetor, que é um software responsável por injetar falhas no sistema, seja antes ou durante a execução. Na técnica de IF os testes são constituídos por dois conjuntos de entrada: a carga de trabalho (*workload*) e a carga de falhas (*faultload*). A carga de trabalho representa as entradas usuais do sistema, que servem para ativar suas funcionalidades, enquanto a carga de falhas representa as falhas a serem introduzidas no sistema.

IF foi proposta com o intuito de testar mecanismos de tolerância a falhas, mas já existem inúmeros trabalhos na literatura propondo o seu uso para testar segurança de aplicações. Thomson et al. (2002) testam a segurança de aplicações injetando falhas durante a execução nas chamadas feitas ao sistema operacional. Outro trabalho utiliza um injetor de falhas para testar *firewalls* e sistemas de detecção de intrusão, simulando ataques ao TCP/IP [Wanner e Weber 2003]. [Morais et al. 2009] usam essa técnica para testar um protocolo de segurança usado na comunicação entre dispositivos móveis e a Internet. Para os testes de segurança de *Web Services*, também já existem inúmeros trabalhos, dentre os quais podemos citar os trabalhos [Antunes e Vieira 2009] e [Mello e Silveira 2011], que também utilizam perturbações nas mensagens SOAP para emular ataques, como na nossa proposta. No entanto, os dois primeiros trabalhos utilizam injetores específicos para o tipo de ataque, enquanto nosso injetor é de propósito geral. Além disso, nos demais trabalhos, somente corrupção de mensagens podem ser utilizados, enquanto, no nosso caso, outros tipos de perturbação podem ser realizados, como por exemplo, atraso na entrega de mensagens.

3. Abordagem proposta

Nesta seção começamos por mostrar a arquitetura de teste utilizada, e em seguida, apresentamos passo-a-passo a realização dos experimentos.

3.1. Arquitetura de testes

A abordagem proposta utiliza IF como técnica para testes de segurança. Utilizamos para esse fim um injetor desenvolvido em um trabalho prévio do grupo chamado WSInject [Valenti et al. 2010]. A ferramenta atua como um Proxy entre um cliente e um servidor, e permite injetar falhas de comunicação tanto para os testes de um serviço quanto nos testes da composição de serviços. No presente estudo, o objetivo foi testar serviços isoladamente; nesse caso o injetor intercepta as requisições enviadas pelo cliente, através de mensagens SOAP, antes destas serem repassadas ao servidor, conforme ilustrado na Figura 2.

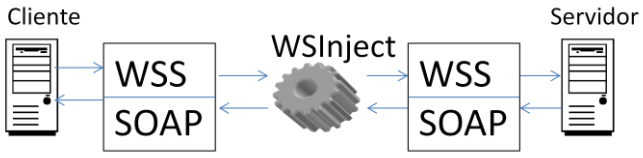

Figura 2. Arquitetura de testes utilizada.

Dado que a WSInject se comporta como um servidor *proxy* HTTP, sua utilização requer pouca instrumentação: basta configurar o cliente para se conectar ao WS alvo através do proxy. A interceptação e modificação das mensagens trocadas entre o cliente e o servidor são transparente para um e outro. Dessa forma, a WSInject não necessita do código fonte do serviço, e nem interfere na plataforma de execução, o que a torna possível de ser utilizada tanto por provedores quanto por usuários do serviço.

A WSInject utiliza scripts para descrição das falhas a serem injetadas [Valenti, Maja, Martins 2010]. Scripts são arquivos de texto contendo um ou mais *FaultInjectionStatements* (comandos de injeção de falhas). Cada *FaultInjectionStatement* é composto de um *ConditionSet* (conjunto de condições) e uma *FaultList* (lista de falhas). Os *FaultInjectionStatements* funcionam como comandos do tipo **condição-ação**: ao interceptar uma mensagem, se esta satisfaz a um conjunto de condições, a lista de falhas é injetada. A Figura 3 mostra um exemplo de script executável pela WSInject. Em negrito temos as palavras-chave para especificar condições e ações. A primeira linha contém uma condição e duas ações, que correspondem às ações que o injetor deve realizar para injetar falhas. Nesse caso, a cada vez que a URI[6] de uma chamada ao Web Service, ou sua resposta, contiver a cadeia "Hotel", substitua todas as ocorrências de "Name" na mensagem por "Age", e duplique o conteúdo da mensagem. Na segunda linha, toda vez que uma mensagem contiver a cadeia "caught exception" e for uma resposta, o conteúdo da mensagem é esvaziado.

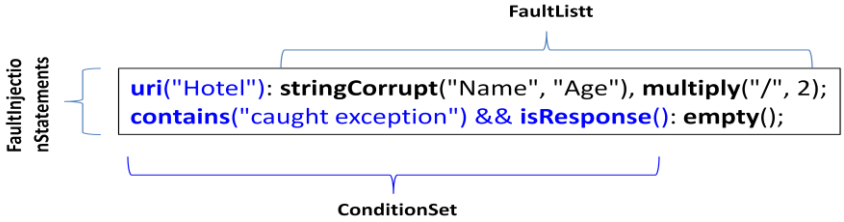

Figura 3. Exemplo de script [Valenti et al. 2010].

[6] Uniform Resource Identifier é uma cadeia de caracteres usada para identificar recursos na Internet.

Uma vez definida a arquitetura de testes, para realizar os ataques foram necessários os seguintes passos:

1. **Preparação**: neste passo foram determinados: Quais ataques realizar? Qual carga de falhas utilizar para emular esses ataques?
2. **Execução**: este passo visa responder às questões: qual carga de trabalho utilizar? Quando injetar as falhas? Que informações coletar durante a execução?
3. **Análise dos resultados**: como determinar se uma vulnerabilidade foi revelada ou não?

Esses passos são explicados mais detalhadamente nas seções a seguir.

3.2.Preparação

Embora tenha sido possível emular diversos tipos de ataques com a WSInject, nesse artigo apresentamos os testes utilizando *XPath Injection*. Conforme mencionado, o atacante, nesse caso, explora comandos XPath (XML Path Language) para atacar servidores que fornecem informação para construir consultas XPath que são usados por usuários para obter dados XML. Através do envio de informações intencionalmente malformadas para o Servidor, um atacante pode conhecer a estrutura dos dados XML e aceder aos dados que são protegidos. O atacante pode até ser capaz de elevar seus privilégios no servidor se os dados XML estão sendo usados para autenticação do cliente. *XPath Injection* pode ter graves consequências, dado que XPath não possui técnicas para controle de acesso e permite a consulta completa dos banco de dados do servidor (documentos XML), permitindo ter acesso até às tabelas de administrador, inacessíveis em consultas regulares [Eviware 2011]. Por exemplo, suponha que o servidor mantenha o arquivo XML mostrado na Figura 4, com informações sobre usuários:

```
<users>
        <user>
         <name>Alice</name>
        <password>ghj348</password>
        <access>Admin</access>
        </user>
        <user>
        ...
</users>
```

Figura 4. Exemplo de trecho de arquivo XML com informações de usuários.

Uma aplicação Web que necessite autenticar um usuário, pede que o cliente forneça seu nome e senha, e gera para o servidor um comando XPath da forma:

```
String xpathQuery = "//user[name/text()='" +
request.get("username") + "' And password/text()='" +
request.get("password") + "']";
```

Um atacante pode fornecer como "*username*" a seguinte cadeia:

```
lol' or 1=1 or 'a'='a
```

Com essa modificação, a autenticação seria ignorada e o atacante conseguiria entrar no sistema.

Para emular esse tipo de ataque, o injetor deve interceptar mensagens SOAP contendo comandos XPath e corromper os valores dos parâmetros. Para definir os valores a serem usados, nos baseamos em informação da literatura, bem como em ataques produzidos por VS, como o soapUI-Security Testing[7]. Exemplos de scripts gerados para esse fim estão mostrados na Tabela 1.

Tabela 1. Exemplos de scripts usados para emular XPathInjection.

Script WSInject	Descrição
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\" or name(//users/LoginID[1]) = 'LoginID' or 'a'='a\">");`	O script usa a condição **isRequest()** que seleciona somente as requisições de um cliente para um servidor. Para cada requisição o injetor realiza **stringCorrupt** para substituir as ocorrências da operação urn:Order Order_ID=\"1\>" por uma consulta XPath (ex.: "<urn:Order Order_ID=\" or name(//users/LoginID[1]) = 'LoginID' or 'a'='a\">") para conhecer a estrutura dos dados XML, e aceder aos dados que são protegidos.
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"' or '1'='1\">");`	
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"1/0\">");`	
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"'%20o/**/r%201/0%20--\">");`	
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"or name(//users/node()) = 'LoginID' or '1='1\">");`	

O WS-Security oferece proteção contra esse tipo de ataque, usando as credenciais de segurança com informações de autenticação (c.f. §2.1). As informações de segurança são inseridas dentro de *tags* `<wsse:Security>`, sendo que cada mensagem SOAP pode conter um ou mais *tags*.

Dado que uma mensagem SOAP pode passar por vários serviços intermediários até atingir o destino final, o WS-Security permite que os serviços intermediários só possam ler ou modificar os trechos de mensagem que lhes são direcionados. A Figura 5 mostra um exemplo de mensagem SOAP com credencial de segurança.

1 <soapenv:Envelope>	Envelope; toda mensagem SOAP deve conter:
2 xmlns:soapenv="..." xmlns:wsse="...">	Declarações contendo, entre outros, como os dados são representados no documento XML
3 <soapenv:Header>	Início de cabeçalho
4 <**wsse:Security**>	Indica início de trecho seguro
5 <**wsse:UsernameToken** wsu:Id="...">	
6 <**wsse:Username**>Alice</**wsse:Username**>	Credencial que representa identificação do cliente <wsse:UsernameToken>
7 <Password Type="PasswordText">Senha</Password>	
8 </**wsse:UsernameToken**>	
9 </**wsse:Security**>	
10 </soapenv:Header>	
11 <soapenv:Body>	Corpo da mensagem que será transmitida ao WS
12 ...	
13 </soapenv:Body>	
14 </soapenv:Envelope>	Termina o conteúdo da mensagem

Figura 5. Exemplo de mensagem SOAP com credencial de segurança (Security Token).

No caso do exemplo, o WS destinatário é informado que o cliente que envia a requisição foi devidamente autenticado, conforme indicado nas linhas 6 e 7 da Tabela 2.

[7] http://www.soapui.org/Security/

Como parte da fase de Preparação, foram feitos experimentos de pré-análise, que nos permitiram, entre outros: **i)** determinar a carga de falhas a ser injetada, o que incluiu a identificação das operações e dos parâmetros a serem injetados; e **ii)** identificar o comportamento do serviço em presença de falhas que caracteriza um ataque bem sucedido. Os resultados dessa pré-análise serão definidos nas próximas seções.

3.3.Execução

Um aspecto importante no Teste de *Web Services* é a geração do tráfego de rede – a carga de trabalho. A carga de trabalho representa as requisições que ativam o WS alvo. Para ser o mais realista possível durante os testes, dever-se-ia gerar tráfego bem próximo do fluxo real de mensagens SOAP. Para gerar a carga de trabalho (*Workload*) foi utilizada a ferramenta soapUI[8], representando o Cliente mostrada na Figura 2. O tráfego gerado consiste em requisições feitas pelo soapUI aos *Web Services* reais com propósito de emular um cliente real fazendo requisições. Para a campanha de injeção foram selecionados 10 *Web Services*, 5 dos quais usam o padrão WS-Security e 5 deles, não.

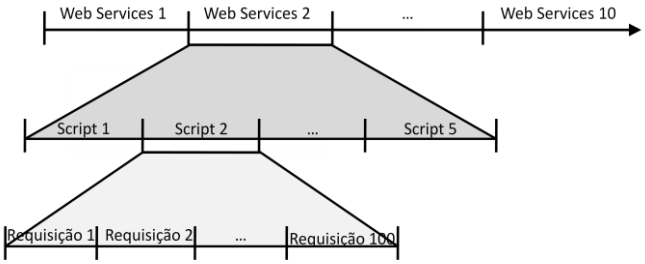

Figura 5. Campanha de Injeção de *XPath Injection*.

A Figura 5 ilustra como foi realizada a campanha de injeções para os serviços testados. Para cada WS foram realizados 5 scripts de injeção, em que cada script especifica a corrupção do valor de um determinado parâmetro, de uma determinada operação, conforme ilustrado na Tabela 1. Para cada script, a carga de trabalho consistiu no envio de 100 requisições. No total, 5000 ataques foram realizados.

Cumpre notar que o número de ataques possíveis de serem realizados, em que cada parâmetro de cada operação de cada serviço seria corrompido com diferentes valores, seria inviável de ser realizado dentro do tempo previsto para a realização dos testes. Por essa razão, optamos por realizar apenas um subconjunto dos testes. Os experimentos de pré-análise nos permitiram determinar as operações e parâmetros que seriam mais interessantes usar como alvo, ou seja, aqueles que levaram a maior ocorrência de defeitos (*failures*) na pré-análise de um conjunto de 69 *Web Services*.

3.4.Análise dos resultados

Um aspecto importante nesse passo é conseguir identificar quando uma vulnerabilidade foi efetivamente detectada, i.e., quando um ataque foi bem sucedido, excluindo potenciais falsos positivos. Devemos diferenciar quando um resultado

[8] http://www.soapui.org/

inválido é obtido devido a uma falha interna (não intencional) do serviço, ou se é consequência de um ataque bem sucedido.

A abordagem proposta é caixa preta, ou seja, não se teve acesso nem ao código fonte dos serviços, e, à exceção de um dos serviços, também não se teve acesso aos respectivos servidores em que residiam. Para determinar se um ataque foi bem sucedido, utilizamos como fontes de informação os logs armazenados pelas ferramentas soapUI e WSInject. Os logs contêm as requisições feitas pelo cliente, bem como a resposta enviada pelo servidor. A figura 6 mostra um exemplo do log produzido pela WSInject, onde estão assinaladas as diferenças tanto na requisição (alterada após a corrupção do parâmetro) quanto da resposta enviada pelo serviço. Como se pode perceber, nesse caso o ataque foi bem sucedido, dado que o serviço não detectou o ataque e respondeu ao cliente.

Script 2
isRequest(): stringCorrupt("<per:PersonID>", "<per:PersonID> ' or '1'='1");

Requisição	Resposta
<soapenv:Body> <per:PersonReq> <per:PersonID> ' or '1'='1Identificador</per:PersonID> </per:PersonReq> </soapenv:Body></soapenv:Envelope>	HTTP/1.1 200 OK // O WS não reconheceu o ataque Content-Type: text/xml;charset=utf-8 Transfer-Encoding: chunked Date: Thu, 08 Mar 2012 18:04:41 GMT <?xml version='1.0' encoding='UTF-8'?> <SOAP-ENV:Envelope xmlns:SOAP-ENV="**http://schemas.xmlsoap.org/soap/envelope/**"> <SOAP-ENV:Body> <PersonRes xmlns:msgns="**http://j2ee.netbeans.org/wsdl/PersonSvc/PersonSvc**" xmlns="**http://xml.netbeans.org/schema/Person**"> <PersonID> ' or '1'='1Identificador</PersonID> <FamilyName>Jose</FamilyName> <GivenName>Martins </GivenName> </PersonRes> </SOAP-ENV:Body> </SOAP-ENV:Envelope>

Figura 6. Exemplo de log gerado pela WSinject.

Além disso foram usados os códigos de status das respostas enviados pelo protocolo HTTP, em especial, o código de erro 500, que indica erros do servidor. Essas respostas foram obtidas analisando-se o log produzido pela WSInject descritos na Tabela 3.

A partir destas informações analisamos as respostas de todos os WS, aplicando um conjunto de regras que nos permitiram identificar quando uma vulnerabilidade foi efetivamente detectada pelos testes, e também quando um ataque foi devidamente rechaçado pelo mecanismo de segurança do WSS, para os serviços que o implementavam. Esse passo é crucial, para reduzir o número de falsos positivos, i.e, para que não fosse indicada uma vulnerabilidade onde não havia. A análise dos resultados se baseou na lista de códigos de status HTTP, descrita na Tabela 4.

Para levantar o conjunto de regras, utilizamos a fase de pré-análise, citada em §3.2, na qual selecionamos um conjunto de 69 serviços. Primeiramente os serviços foram executados sem injeção de falhas, para se determinar as respostas fornecidas. Em seguida, foram introduzidos ataques, utilizando um VS – soapUI-Security Test.

Tabela 3. Lista de Códigos de status HTTP.

Códigos HTTP	Descrição
200 – OK	Padrão de resposta para solicitações HTTP bem sucedida. Vulnerabilidade confirmada porque o sistema executou a requisição sem detectar o ataque.
400 – Bad Request	A requisição não pode ser atendida devido à sintaxe ruim. Consideramos como ataque mal sucedido dado que o servidor detectou o ataque.
500 – Internal Server Error	O servidor não cumpriu com uma solicitação aparentemente válida ou encontrou uma condição inesperada que o impediu de atender a requisição feita pelo cliente. Consideramos analisar a resposta do servidor usando o *tag* <soap:Fault> dentro do corpo (body) da mensagem; essa *tag* fornece os erros e a informação de status da mensagem SOAP, contendo os sub-elementos: • <faultcode> Codigo de identificação da falha. • <faultstring> Explicação legível da falha. • <faulttactor> Informação de quem/que fiz acontecer a falha. • <details> Informação detalhada do erro. Os valores de faulcode podem ser classificados em 4 tipos: • VersionMismatch: O servidor encontrou um espaço de nome (*namespace*) inválido no envelope da mensagem SOAP. • MustUnderstand[9]: A falha de MustUnderstand indica a ausência do um elemento obrigatório no cabeçalho da mensagem SOAP. • Client: A mensagem foi estruturado de forma incorreta ou contém informações incorretas. • Server: Aconteceu um problema com o servidor de forma que a mensagem não possa ser processada.

Com base nos resultados obtidos na fase de pré-análise, e na interpretação dos códigos de status HTTP dado na Tabela 3, criamos um conjunto de regras para analisar os resultados, de forma análoga ao trabalho de Antunes e Vieira (2009), obtendo um conjunto de 11 regras, descritas na Tabela 4.

Os resultados dos ataques para os 10 *Web Services* estão resumidos na Tabela 5. Pode-se notar que todos os ataques especificados puderam ser injetados pela WSInject. De resto, como seria de se esperar, a maioria dos ataques puderam ser detectados pelos serviços que usam WSS. Essa diferença ocorre porque as **credenciais de segurança** verificam a autenticidade do cliente, além de outros parâmetros.

Os 28% de ataques bem sucedidos correspondem a dois serviços que o atacante conseguiu explorar vulnerabilidades no sistema. Um dos WS retornou o código 200 em resposta a requisições corrompidas, ou seja, o serviço não conseguiu detectar o ataque, fornecendo ao atacante a resposta que este esperava (c.f. regra 2 na Tabela 4). O outro reconheceu parcialmente o ataque devolvendo o erro 500 indicando problemas de sintaxes na mensagem SOAP. No entanto, forneceu informações internas detalhadas, que podem servir para outros ataques (c.f. Regra 7 na Tabela 4).

[9] O atributo MustUnderstand serve para indicar se uma entrada no cabeçalho é obrigatório ou opcional.

Tabela 4. Regras para determinar ataques bem ou mal sucedidos em *Web Services*

Regras	Descrição
Regra 1	Se a resposta enviada contém mensagem de tipo "*Ok*" o código "*HTTP/1.1 200 OK*" (ex.: ver o log gerado pelo WSInject na Figura 6), então foi detectada uma vulnerabilidade porque o sistema executa a requisição sem detectar o ataque.
Regra 2	Se a resposta enviada contém mensagem de erro do tipo "bad request message" ou o código "HTTP/1.1 400" (ex.: "Request format is invalid: Missing required soap: Body element."), então não foi detectada vulnerabilidade, pois o servidor foi capaz de detectar a mensagem inválida enviada pelo atacante.
Regra 3	Se a resposta enviada pelo serviço é errônea, tanto na ausência quanto na presença de ataques então não foi detectada vulnerabilidade, ou seja, a resposta errônea não foi causada pelo ataque, mas se deve à existência de outra falha de software.
Regra 4	Se a resposta é válida em presença do ataque e na execução sem falhas houve erro, então foi detectada uma vulnerabilidade, pois o ataque permitiu que mecanismos de autenticação fossem ignorados, ou o acesso a informações sem a devida autorização.
Regra 5	Se a resposta enviada é correta e não houve divulgação de informações sobre execução no servidor (ex.: stack trace), que permita ao atacante tomar conhecimento sobre o software usado no servidor, então não foi detectada vulnerabilidade.
Regra 6	Se a resposta enviada contém exceções geradas pelo analisador XPath, então foi detectada uma vulnerabilidade.
Regra 7	Se a resposta retornada pelo serviço contém informações sensíveis, tais como divulgação do cookie da sessão do usuário (seqüestro de sessão), ou exibe informações detalhadas sobre a conexão do usuário (método de criptografia, o uso de SSL) ou rotas de diretórios do servidor, então foi detectada vulnerabilidade.
Regra 8	Se o servidor retorna código executado na mensagem SOAP ou redireciona o usuário para outra página ou ainda o servidor não responde, então foi detectada vulnerabilidade.
Regra 9	Se a resposta contém exceções geradas pelo servidor de banco de dados, então foi detectada vulnerabilidade, pois o ataque permitiu executar partes do serviço que não o haviam sido quando entradas válidas foram fornecidas.
Regra 10	Se a resposta errônea foi propagada para outras camadas, gerando códigos de erro ou exceções, então foi detectada vulnerabilidade, pela mesma razão dada acima.
Regra 11	Se nenhuma das regras acima pode ser aplicada, então o resultado é tido como inconclusivo, pois não há forma de confirmar se realmente existe uma vulnerabilidade.

Tabela 5. Sumário dos ataques realizados.

***Xpath Injection* – sem WSS**			
	Total	**Ataques Mal Sucedidos**	**Ataques bem Sucedidos**
Total de ataques desejados	2500	0	2500
% Ataques injetados	100%	0%	100%
Total de scripts	25	0	25
***Xpath Injection* com WSS**			
Total de ataques desejados	2500	1800	700
% Ataques injetados	100%	72%	28%
Total de scripts	25	18	7

Tabela 6. Resultados do ataque de *XPath Injection*.

Web Services	**qun**	**min**	**max**	**média**	**tps**	**bytes**	**bps**
Ataque sem WSS	2500	495	3788	715.586	1.329	2432100	1293.2
Ataque com WSS	2500	8	26524	901.682	1.464	2971416	1352.8
Total	**5000**	**8**	**26524**	**808.634**	**1.397**	**5403516**	**1323**

A Tabela 6 resume o desempenho da execução dos scripts, fornecendo as seguintes informações: quantidade de requisições, tempo mínimo de resposta a uma requisição (ms), tempo máximo de resposta a uma requisição (ms), tempo médio de

resposta a uma requisição (ms), transações ou requisição por segundo, quantidade de bytes enviados e número de bytes por segundo. Podemos observar que as requisições feitas a WS com WSS precisam de mais tempo para ser enviadas (média) e enviam maior quantidade de bytes, como já seria de se esperar.

4. Resultados e Trabalhos Futuros

Nesse artigo mostramos uma abordagem para testes de segurança de *Web Services* usando um injetor de falhas para emular ataques. A título de exemplo, mostramos a emulação do ataque *XPath Injection*, em que um atacante modifica consultas a documentos XML feitas com a linguagem Xpath. Trata-se de um ataque bastante frequente, segundo estudos citados, e cujos efeitos podem ser bastante perigosos para o sistema. Os organismos de padronização propuseram a especificação do WS-Security (WSS), que, entre outros mecanismos, usa credenciais de segurança (*security tokens*) para proteger as mensagens trocadas entre serviços contra esse tipo de ataque. Foram testados 10 serviços, sendo que metade com a implementação desse protocolo, e metade não. Os resultados mostraram a eficácia do WSS na proteção contra esse tipo de ataque.

Uma vantagem da abordagem proposta é que ela se baseia no uso de um injetor de falhas de propósito geral, o qual pode ser usado para emular diversos tipos de ataques, inclusive podendo gerar variantes dos mesmos, o que geralmente é limitado nas ferramentas comumente usadas para testar segurança, como os *vulnerability scanners*. Para mostrar a flexibilidade da ferramenta, foram emulados diversos tipos de ataques. Os resultados não foram mostrados aqui por questão de espaço.

Como trabalho futuro, pretendemos utilizar variantes de ataques para melhorar na detecção de novas vulnerabilidades, sempre considerando o serviço como caixa preta.

5. References

Ana, C. V. de Melo; Silveira, P.; “Improving data perturbation testing techniques for Web services”; Inf. Sci. 181, 3 (February 2011), 600-619.

Antunes, N.; Vieira, M.; "Detecting SQL Injection Vulnerabilities in Web Services". Dependable Computing, 2009.LADC '09. Fourth Latin-American Symposium 2009.

Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J-C.; Laprie, J-C, Martins, E. "Fault injection for dependability validation: a methodology and some applications", 1990.

Avizienis, A.; Laprie, J-C.; Randell, B.; “Dependability and Ist Threats: A Taxonomy”; in: Jacquart, R.l eds.; Proceedings of the IFIP 18th World Computer Congress; 2004.

Cachin, C.; Camenisch, J.; “Malicious and Accidental-Fault Tolerance in Internet Applications: Reference Model and Use Cases”, LAAS, MAFTIA, 2000.

Eastlake, D.; Reagle, J.; Imamura, T.; Dillaway, B.; Simon, E; “XML Encryption Syntax and Processing”. W3C Recommendation. 10/Dec/2002.

Eastlake, D.; Reagle, J.; Solo, D.; Hirsch, F.; Roessler, T.; Bartel, M.; Boyer, J.; Fox, B.; LaMacchia, B.; Simon. “XML Signature Syntax and Processing, 2nd Edition”. 2008.

Eviware. soapUI; the Web Services Testing tool – Security Testing Tool.

Garcia, R. "Case study: Experiences on SQL language fuzz testing", DBTest 09 Proceedings of the Second International Workshop on Testing Database Systems.

Hartman, B.; Flinn, D.; Beznosov, K.; Kawamoto, S.. “Mastering Web Services Security”, Wiley Publishing, Inc, ISBN-13: 978-0471267164. Jan. 2003.

Holgersson, J.; Soderstrom, E. “Web Wervice Security-Vulnerabilities and Threats Within the Context of WS-Security”. SIIT 2005, ITU, Geneva, Sep/2005.

IBM Corp.; Microsoft Corp. Whitepaper “Security in a Web Services World A Proposed Architecture and Roadmap”. April 7, 2002, V1.0.

Lawrence, K.; Kaler, C.; Nadalin, A.; Monzillo, R.; Hallam-Baker, P. “Web Services Security: UsernameToken Profile 1.1”. OASIS Standard Specification. 2006-A.

Lawrence, K.; Kaler, C.; Nadalin, A.; Monzillo, R.; Hallam-Baker, P. “Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)”. OASIS, 2006-B.

Mello, E. R.; Wangham, M. S.; Fraga, J. S.; and Camargo, E. S. (2006). Segurança em serviços web. In Minicursos do SBSeg 2006, Santos, SP.

Miller, B. et al.; “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, Computer Sciences Technical Report #1268, 1995.

Moorsel, A. V.; Bondavalli, A.; Pinter, G.; Madeira, H.; Majzik, I.; Durães, J.; Karlsson, J.; Falai, L.; Strigini, L.; Vieira, M.; Vadursi, M.; Lollini, P. Esposito, R. “State of the Art, AMBER - Assessing, Measuring, and Benchmarking Resilience”- report, 2009.

Morais, A.; Martins, E.; “Injeção de Ataques Baseados em Modelo para Teste de Protocolos de Segurança”. Dissertação (Mestrado em Ciências da Computação) – Instituto de Computação, Universidade Estadual de Campinas. 15/Maio/2009.

Morais, A.; Martins, E.; and Cavalli, A. 2009. Security Protocol Testing Using Attack Trees. In Proceedings of 2009 International Conference on Computational Science and Engineering, (Vancouver, Canada, Aug. 2009).

Singhal, A.; Winograd, T.; Scardfone, K.; “Guide to Secure Web Services”; Recommendations of the National Institute of Standards and Technology (NIST).

Valenti, A. W.; Maja, W. Y.; Martins, E.; Bessayah, F.; and Cavalli, A. “WSInject: A Fault Injection Tool for Web Services".Technical Report.ICa10a22, Campinas, "Brazil: Instituto de Computação, Universidade Estadual de Campinas, July 2010.

Vieira, M.; Antunes, N.; Madeira, H. "Using Web Security Scanners to Detect Vulnerabilities in Web Services".Conf. on Dependable Systems and Networks, 2009.

Wanner, P.C.H. and Weber, R.F. 2003. Fault Injection Tool for Network Security Evaluation. LNCS, vol. 2847/2003, Dependable Computing. (Sep. 2003).

Zhao, G.; Zheng, W.; Zhao, J.; Chen, H. "An Heuristic Method for Web-Service Program Security Testing," ChinaGrid Annual Conference, 2009.