Usernames & Passwords disclosure

Autor: Guillermo Hernández
Fecha: 22 de junio del 2017
Versión 1.0
Nivel de acceso: Desclasificado con reservas (se omite nombre de la app y empresa afectadas, además de nombres de usuario).

Introducción

Dentro de las labores de investigación que llevo a cabo, me fue encomendado hacer un análisis de una aplicación móvil para pago de peajes con TAG, para intentar acceder a la URL de un contrato WSDL.

Además de lo solicitado, encontré una puerta trasera en la aplicación Web que se usa para administrar los mensajes y avisos que son mostrados en las aplicaciones móviles.

En este artículo, utilicé el siguiente ambiente de desarrollo/pruebas: Virtual Box ejecutando un Windows XP SP3 y VB6. PC con Ubuntu 16 y las siguientes herramientas: NMAP, Dex2Jar, Python. Otro PC con Windows 10, Navegador TOR y Java Decompiler (http://jd.benow.ca/). Un teléfono Android con la aplicación involucrada y APK Extractor.

El primer análisis

Según la cuenta de Twitter de dicha aplicación: *… es un método de pago seguro y confiable para las autopistas urbanas y las carreteras más importantes del país*.

Dicha empresa tiene una aplicación Android que permite ver información de tarifas, mapas, ponerse en contacto con su área de Atención al Cliente y hacer recargas al tag.

Los textos que muestra la aplicación, incluidos los costos de los trayectos, son obtenidos de una Aplicación Web administrable mediante una especie de mini CMS (Content Managment System) desarrollado a medida.

Puesto que lo que se quería obtener era la URL de un servicio Web el primer paso fue extraer el APK de la aplicación para desensamblarlo con Dex2Jar y así obtener el código fuente.

Para acceder al código fuente del APK primero se debe convertir el archivo DEX (“nativo de Android” – Dalvik Executable) a JAR, así que lo extraje con APK Extractor. Después en una terminal en la PC con Windows 10:

```
C:\dex2jar-2.0>ren vulnerableAppDownloaded.apk Hackme.apk

C:\dex2jar-2.0>mkdir hackme

C:\dex2jar-2.0>d2j-dex2jar -f -o hackme/app.jar HackMe.apk

C:\dex2jar-2.0>cd hackme

C:\dex2jar-2.0>dir
```

Posteriormente abrí el archivo app.jar con JD-GUI

Encontrando los Webservices

Para encontrar los servicios Web, procedí a buscar URL *hardcodeadas* dentro del código fuente de la aplicación. Las URL's a buscar eran aquellas con el formato típico de un Webservice, es decir, aquellas con terminación ***?wsdl***.

No se encontró ninguna URL que pudiera darme información acerca del contrato, sin embargo, sí encontré prácticamente todo el código Java que genera el XML que se usa para consumir tales servicios.

Estructura de los XML que se generan

A continuación se describen las rutinas en Java que generan XML del Webservice de recarga a TAGs de prepago.

Campos obligatorios:

- Amount
- User
- Id_order
- Id_tag
- Password

Para generar el XML se vale de los métodos *startDocument*, *startTag* y *text* del paquete org.xmlpull.v1, el cual se puede descargar de Internet.

```
  public String createXMLString(String paramString1, String paramString2, String
paramString3)
  {
    try
    {
      XmlSerializer localXmlSerializer = Xml.newSerializer();
      StringWriter localStringWriter = new StringWriter();
      localXmlSerializer.setOutput(localStringWriter);
      localXmlSerializer.startDocument("UTF-8", Boolean.valueOf(true));
      localXmlSerializer.startTag("", "topup_service");
      localXmlSerializer.startTag("", "request");
      localXmlSerializer.startTag("", "user");
      localXmlSerializer.text("WSUser");
      localXmlSerializer.endTag("", "user");
      localXmlSerializer.startTag("", "password");
      localXmlSerializer.text("WSPa55w0rd");
      localXmlSerializer.endTag("", "password");
      localXmlSerializer.startTag("", "id_tag");
      localXmlSerializer.text(paramString1);
      localXmlSerializer.endTag("", "id_tag");
      localXmlSerializer.startTag("", "id_order");
      localXmlSerializer.text(paramString2);
      localXmlSerializer.endTag("", "id_order");
      localXmlSerializer.startTag("", "amount");
      localXmlSerializer.text(paramString3);
      localXmlSerializer.endTag("", "amount");
      createAttributesStringArrayForFieldsRequired();
      localXmlSerializer.endTag("", "request");
      localXmlSerializer.endTag("", "topup_service");
      localXmlSerializer.endDocument();
      paramString1 = localStringWriter.toString();
      return paramString1;
    }
    catch (Exception paramString1)
    {
      paramString1.printStackTrace();
    }
    return null;
  }
```

Dentro de este código, también se aprecia el usuario y contraseña del servicio Web

```
      localXmlSerializer.startTag("", "user");
      localXmlSerializer.text("WSUser");
      localXmlSerializer.endTag("", "user");
      localXmlSerializer.startTag("", "password");
      localXmlSerializer.text("WSPa55w0rd");
      localXmlSerializer.endTag("", "password");
```

El XML generado debería de lucir similar a este:

```
<?xml version="1.0" encoding="utf-8"?>
<toupservice>
	<request>
		<user>WSUser</user>
		<password>WSPa55w0rd</password>
		<id_tag>texto</id_tag>
		<id_order>texto</id_order>
		<amount>numero</amount>
	</request>
</toupservice>
```

El valor de los campos id_tag, id_order y amount son obtenidos del formulario de recarga de la app. Desafortunadamente no tuve acceso al mismo dentro de la App, ya que requiere un TAG para crear una cuenta.

Por su parte la respuesta del webservice consiste en otro XML en el que se indica el código de respuesta y el nuevo monto (“saldo”) del TAG.

```
  public String createXMLString(String paramString1, String paramString2, String
paramString3)
  {
    try
    {
      paramString1 = Xml.newSerializer();
      paramString2 = new StringWriter();
      paramString1.setOutput(paramString2);
      paramString1.startDocument("UTF-8", Boolean.valueOf(true));
      paramString1.startTag("", "topup_service");
      paramString1.startTag("", "response");
      createAttributesStringArrayForFieldsRequired();
      paramString1.startTag("", "balance");
      paramString1.text(getBalance());
      paramString1.endTag("", "balance");
      paramString1.startTag("", "ret_code");
      paramString1.text(getRet_code());
      paramString1.endTag("", "ret_code");
      paramString1.startTag("", "ret_msg");
      paramString1.text(getRet_msg());
      paramString1.endTag("", "ret_msg");
      paramString1.endTag("", "response");
      paramString1.endTag("", "topup_service");
      paramString1.endDocument();
      paramString1 = paramString2.toString();
      return paramString1;
    }
    catch (Exception paramString1)
    {
      paramString1.printStackTrace();
    }
    return null;
  }
```

El XML de respuesta tendría que ser parecido a este:

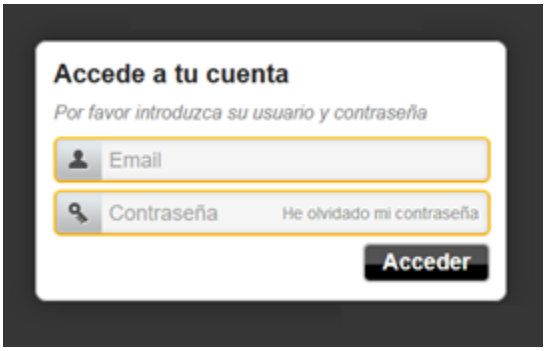
```
<?xml version="1.0" encoding="utf-8"?>
<toupservice>
	<response>
		<balance_service>
			<request>
				<user>WSUser</user>
				<password>WSPa55w0rd</password>
				<id_tag>texto (numero de tag)</id_tag>
			</request>
		</balance_service>
		<balance>nuevo monto o saldo</balance>
		<ret_code>codigo de respuesta </ret_code>
		<ret_msg>Texto de respuesta</ret_msg>
		</response>
</toupservice>
```

Encontrando la URL de la WebApp

En mi búsqueda de URLs que pudieran ser del contrato WSDL del Webservice, encontré un fragmento de código como este:

```
new String[] { String.format("http://19X.XX.XXX.XX:82/api/%s"
```

De manera prácticamente automática introduje la IP y el puerto en la barra de direcciones del navegador TOR, el resultado fue que obtuve la página del login de una Aplicación Web:



Dentro del código empecé a buscar indicios de lo que pudieran ser rutas de acceso, para ponerlas después de la URL encontrada. El formato de la misma, al terminar con `/api/%s`, me hizo saber que `%s` debería ser reemplazado por algún otro método o parámetro para consumir un servicio al estilo REST.

Dicha búsqueda dio resultados cuando encontré lo siguiente:

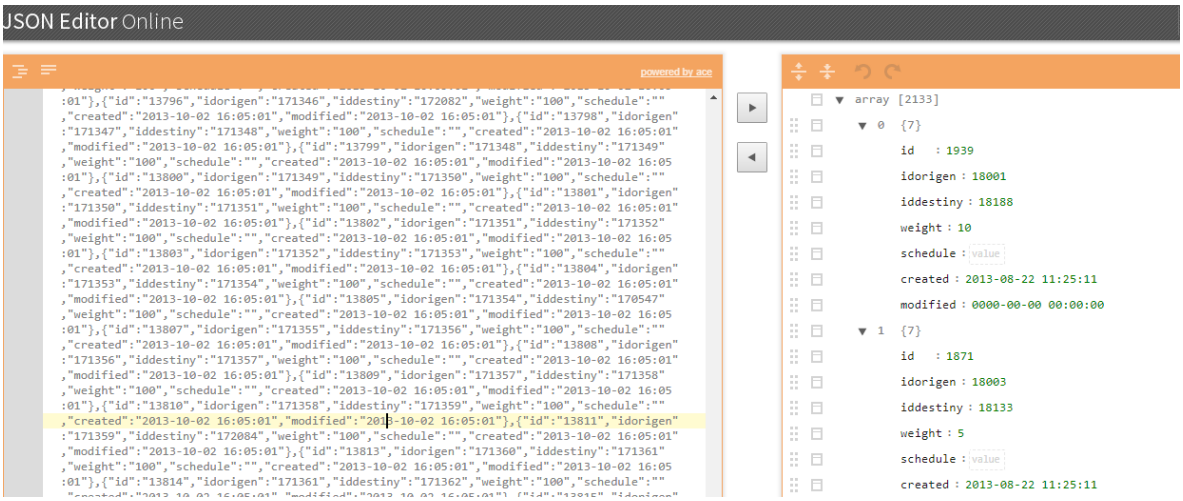
```
new DownloadRoadAsyncTask(3.0F * getResources().getDisplayMetrics().density,
this.mOnFinishDownloadPolylineListener).execute(
new String[] {
String.format("http://19X.XX.XXX.XX:82/api/%s", new Object[] { "highways" }),
String.format("http://19 X.XX.XXX.XX:82/api/%s", new Object[] { "highwaycombinations" }),
String.format("http://19 X.XX.XXX.XX:82/api/%s", new Object[] { "highwayrates/" })
});
```

Explotando su API

Con la información recolectada ahora podía construir URL completas que consumían lo que parece un servicio REST

- http://19X.XX.XXX.XX:82/api/highways (carreteras donde se acepta el pago con TAG)
- http://19 X.XX.XXX.XX:82/api/highwaycombinations (parecen entronques para poder cambiar de carretera, una especie de "correspondencia")
- http://19 X.XX.XXX.XX:82/api/highwayrates/ (precios de los peajes)

Todos los servicios retornan un objeto JSON que puede ser analizado fácilmente con un editor, por ejemplo http://jsoneditoronline.org/



Una búsqueda más detallada me llevo a obtener los siguientes métodos (todos después de /api)

- /users
- /user/
- /highways
- /highwaycombinations
- /highwaysrates
- /ads
- /activations
- /billings
- /cacs
- /followus
- /promotions
- /rates
- /refills
- /socialmessage
- /socialmessages
- /tags
- /tracks

Consumir la parte de usuarios

De los métodos encontrados el que nos interesa es /users, ya que responde con un JSON como el siguiente:


```
[
	{
		"id": "45",
		"group": "2",
		"name": "Javier",
		"username": "jcXXXo@ohXXXXXXXXXXXes.com",
		"password": "aaa19410b46995311f6e27ba23b2479f35206be9",
		"email": "jcXXXo@ohXXXXXXXXes.com",
		"created": "2014-08-29 06:47:28",
		"modified": "2016-01-07 15:58:03",
		"permission": "101111111111111"
	},
	{
		"id": "44",
		"group": "2",
		"name": "Da___ _____ra",
		"username": "dyXXXa@oXXXm.com.mx",
		"password": "20c2c3a7403c2d24cf0408466945aa4c91d267a9",
		"email": "dyXXXa@oXXXm.com.mx",
		"created": "2014-08-27 16:28:41",
		"modified": "2014-08-27 16:28:41",
		"permission": "101111111111111"
	}
]
```


También podemos obtener los datos de un usuario en particular si usamos el API de esta forma:

`/user/<numero>`

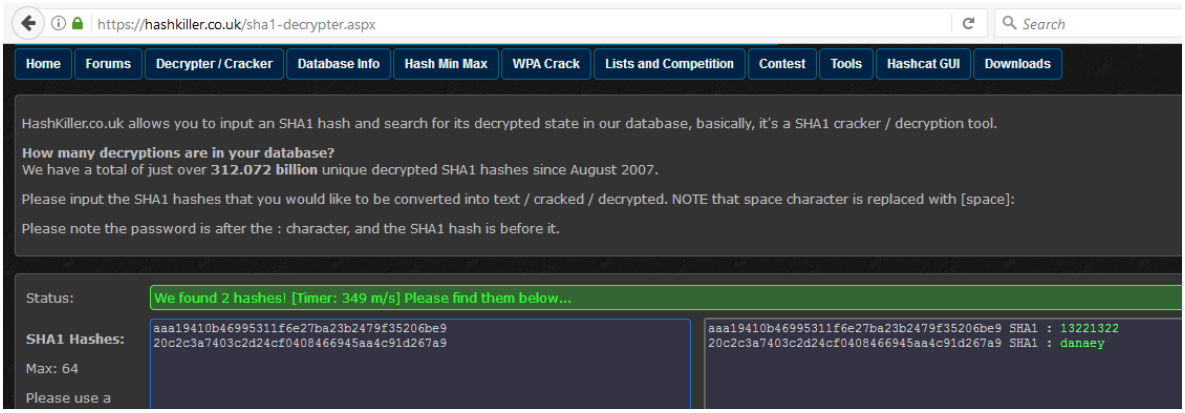
Por ejemplo `/api/user/45`, y nos aparecen los datos de Javier

Bypass al SHA1 de las contraseñas

En las respuestas JSON anteriores se puede apreciar un digest en SHA1 con las contraseñas de cada usuario. Si se buscan los digests en un diccionario, podemos obtener la contraseña en claro de ambos. Para este artículo utilicé el diccionario de http://hashkiller.co.uk

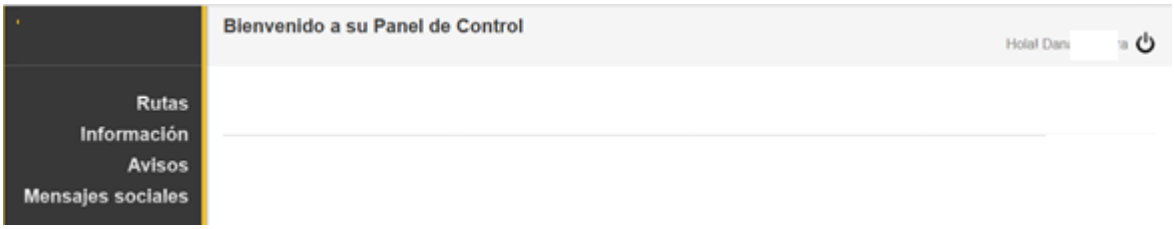
Se encontraron sin problema alguno las contraseñas en claro de los 2 usuarios

Usuario	Pass SHA1	Pass texto plano
Javier	aaa19410b46995311f6e27ba23b2479f35206be9	13221322
Da___ _____a	20c2c3a7403c2d24cf0408466945aa4c91d267a9	danaey



Inicio de sesión

Para terminar de explotar la puerta trasera recién encontrada, solo hacía falta iniciar sesión, de esta forma descartaríamos que se tratase de un *honeypot*.



Validando la puerta trasera: Cambiando los textos de la app.

Para validar que la aplicación Web es la que está liberada en su ambiente de Producción decidí cambiar un texto de la Aplicación móvil. En el apartado "*Texto*" se le añadió otro punto después de "*metropolitana*".

Antes	Después
Título Contrátalo Texto Acude a cualquiera de los más de 300 puntos de venta del área metropolitana. Si tu vehículo es blindado acude a un Centro de Atención a Clientes o llama al 5265-8855 y viaja por las autopistas más importantes del país..	Se ha guardado el registro correctamente Título Contrátalo Texto Acude a cualquiera de los más de 300 puntos de venta del área metropolitana.. Si tu vehículo es blindado acude a un Centro de Atención a Clientes o llama al 5265-8855 y viaja por las autopistas más importantes del país..

Así se ve el cambio en la aplicación

Antes	Después
	

Conclusión y recomendaciones

El descubrimiento de esta puerta trasera deja en claro que muchas grandes empresas todavía no tienen una cultura correcta en temas de seguridad al delegar el desarrollo de aplicaciones a programadores poco experimentados y/o no realizar pruebas de penetración en las mismas.

Por su naturaleza, las aplicaciones Web, son los objetivos más buscados por los atacantes.

En este caso en particular, supongamos que, quien descubrió la puerta trasera tuviera malas intenciones, lo que tendría que empezar a hacer comenzar a realizar intentos de otro tipo de ataques, tales como SQL Injection. Incluso puede que sea relativamente fácil subir archivos con código malintencionado, ya que la aplicación permite subir imágenes que después son mostradas en otra parte de la misma (app). En el artículo titulado "*Código malintencionado dentro de una imagen GIF*" describo como llevar a cabo tal procedimiento.

La principal recomendación que puedo hacer en este caso es, nunca enviar las contraseñas al cliente de un servicio REST, (ni de ningún otro). Si por alguna razón es estrictamente necesario enviar contraseñas en la respuesta del servicio, el desarrollador deberá cambiar la arquitectura de la misma para añadir una capa adicional de seguridad a la contraseña enviada. Usar doble hasheo puede ser una buena opción.

Esta puerta trasera fue reportada al CERT de la UNAM, sin obtener respuesta.

Referencias

Referencias y documentación usados en este artículo:

- Qué es un archivo DEX http://www.openthefile.net/es/extension/dex
- Qué es un archivo JAR https://en.wikipedia.org/wiki/JAR_(file_format)
- Web Service Descriptor Language https://es.wikipedia.org/wiki/WSDL
- XML Serializer XMLPull V1 http://www.xmlpull.org/v1/download/
- Código malintencionado dentro de una imagen GIF https://underdog1987.wordpress.com/2015/09/15/codigo-dentro-de-una-imagen-gif/

- Cifrar contraseñas ¿Es totalmente seguro SHA1? https://underdog1987.wordpress.com/2017/02/14/cifrar-contrasenas-es-totalmente-seguro-sha1/

Guillermo Hernandez

Twitter.com/underdog1987

Ha trabajado en el desarrollo de software desde el año 2004, realizando proyectos en diferentes empresas. Se ha desempeñado como instructor de computación impartiendo materias como: arquitectura de computadoras, Excel avanzado y programación en Visual Basic.
Se inició en la programación web en el año 2005 usando ASP, posteriormente incursionó en PHP. Realiza trabajos de investigación de manera independiente en materia de ciberseguridad.