Checkpoint 2

p7zip

Arnav Nidumolu, Atharva Kale, Pascal von Fintel, Patrick Negus

2023-04-06

Checkpoint 2

Contents:

Github Link

https://github.com/atharvakale343/390r-debugging-setup

Dynamic Analysis

Fuzzing

Fuzzing was the main dynamic analysis technique we used against our target `p7zip`. We mainly fuzzed the extract (`e`) feature of our binary as the feature uses several decompression algorithms as part of its execution.

We used `afl-plus-plus` as the primary fuzzing tool.

https://github.com/AFLplusplus/AFLplusplus

Generating a corpus

We took a variety of steps to find a good enough corpus for our fuzzing efforts. The major approach here to was to search online for commonly used corpora. We wanted to find not only `.zip` format, but also as many different formats possible.

We found a decent corpus at https://github.com/strongcourage/fuzzing-corpus

This included the following formats:

- `.zip`
- `.gzip`
- `.lrzip`
- `.jar`

We added this as a target to our fuzzing Makefile.

```
get-inputs:
    rm -rf in_raw fuzzing-corpus && mkdir in_raw

    git clone -n --depth=1 --filter=tree:0 git@github.com:strongcourage/fuzzing-corpus.git
    cd fuzzing-corpus && git sparse-checkout set --no-cone zip gzip/go-fuzz lrzip jar && git checkout
    mv fuzzing-corpus/zip/go-fuzz/* in_raw
    mv fuzzing-corpus/jar/* in_raw
    mv fuzzing-corpus/gzip/go-fuzz/* in_raw
    mv fuzzing-corpus/lrzip/* in_raw
```

The next step was to choose only "interesting" inputs from this corpus. This includes small inputs that don't crash that binary immediately.

We used the `afl-cmin` functionality to minimize the corpus.

```
afl-cmin -i in_raw -o in_unique -- $(BIN_AFL) e -y @@
```

Another important minimization step included `tmin`. This augments each input such that it can be as small as possible without compromising it's ability to mutate and produce coverage in the instrumented target.

Unfortunately, this process takes a long time, and it only completed for us after a day.

```
cd in_unique; for i in *; do afl-tmin -i "$$i" -o "../in/$$i" -- ../$(
    BIN_AFL) e -y @@; done
```

The cybersec room servers come in handy here!

Experimenting with fuzzing composition flags

We discovered that it is not enough to fuzz a plain instrumented target with `afl-plus-plus`. The target binary may not be easily crashed with mutated inputs as `p7zip` has a robust input error checker. We took to fuzzing with various sanitizers instead to search for harder to find bugs.

We used the following sanitizers on our target:

- ASAN: Address Sanitizer: discovers memory error vulnerabilities such as use-after-free, heap/buffer overflows, initialization order bugs etc.
- MSAN: Memory Sanitizer: mainly used to discover reads to uninitialized memory such as structs etc.
- TSAN: Thread Sanitizer: finds race conditions

```
afl:
    rm -rf $(BIN_AFL)
    git clone $(GH_URL) $(BIN_AFL)
    cp 7zz-makefiles/$(BIN_DEFAULT).mak $(BIN_AFL)/CPP/7zip/7zip_gcc.
        mak
    cd $(BIN_AFL)/CPP/7zip/Bundles/Alone2 && CC=$(AFL_CC) CXX=$(AFL_CXX
        ) make -f makefile.gcc

afl-asan:
    rm -rf $(BIN_AFL_ASAN)
    git clone $(GH_URL) $(BIN_AFL_ASAN)
    cp 7zz-makefiles/$(BIN_AFL_ASAN).mak $(BIN_AFL_ASAN)/CPP/7zip/7
        zip_gcc.mak
```

```
    cd $(BIN_AFL_ASAN)/CPP/7zip/Bundles/Alone2 && AFL_USE_ASAN=1 CC=$(
        AFL_CC) CXX=$(AFL_CXX) make -f makefile.gcc

afl-msan:
    rm -rf $(BIN_AFL_MSAN)
    git clone $(GH_URL) $(BIN_AFL_MSAN)
    cp 7zz-makefiles/$(BIN_AFL_MSAN).mak $(BIN_AFL_MSAN)/CPP/7zip/7
        zip_gcc.mak
    cd $(BIN_AFL_MSAN)/CPP/7zip/Bundles/Alone2 && AFL_CC_COMPILER=LLVM
        AFL_USE_MSAN=1 CC=$(AFL_CC) CXX=$(AFL_CXX) make -f makefile.gcc

afl-tsan:
    rm -rf $(BIN_AFL_TSAN)
    git clone $(GH_URL) $(BIN_AFL_TSAN)
    cp 7zz-makefiles/$(BIN_AFL_TSAN).mak $(BIN_AFL_TSAN)/CPP/7zip/7
        zip_gcc.mak
    cd $(BIN_AFL_TSAN)/CPP/7zip/Bundles/Alone2 && AFL_USE_TSAN=1 CC=$(
        AFL_CC) CXX=$(AFL_CXX) make -f makefile.gcc
```

Parallel Fuzzing

To start with, our approach was to fuzz the `extract` command of `7zz`. So we found an appropriate corpus and fuzzed with the `e` command-line argument (along with `-y` to account for same filenames / avoid user input hangs).

With all different sets of compilation flags that we mentioned previously, we compiled the binaries with AFL instrumentation. Then, to more effectively fuzz, we setup a parallel fuzzing environment in one of the **CyberSec club** VMs.

We added the `afl-fuzz` commands in a `Makefile` and followed the official guide for using multiple cores. Below are the commands we utilized. All of our fuzzers shared the same input and output directores to keep track of current fuzzing state.

```
AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 $(AFL_FUZZ)
    -M main-afl-$(HOSTNAME) -t 2000 -i in -o out -- $(BIN_AFL) e -y @@
```

Our main fuzzer used a regular instrumented AFL binary with no other `CFLAGS`. We used a timeout of 2 seconds to denote a hang (or infinite loops).

```
AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 $(AFL_FUZZ)
    -S variant-afl-asan -t 2000 -i in -o out -- $(BIN_AFL_ASAN) e -y @@
```

Our variant fuzzers utilized binaries compiled with other flags (such as *asan* and *msan*). These had the same timeout as before of 2 seconds.

To keep track of all fuzzers and run them simultaneouly, we used `tmux` sessions with a separate window for each fuzzer.

Results

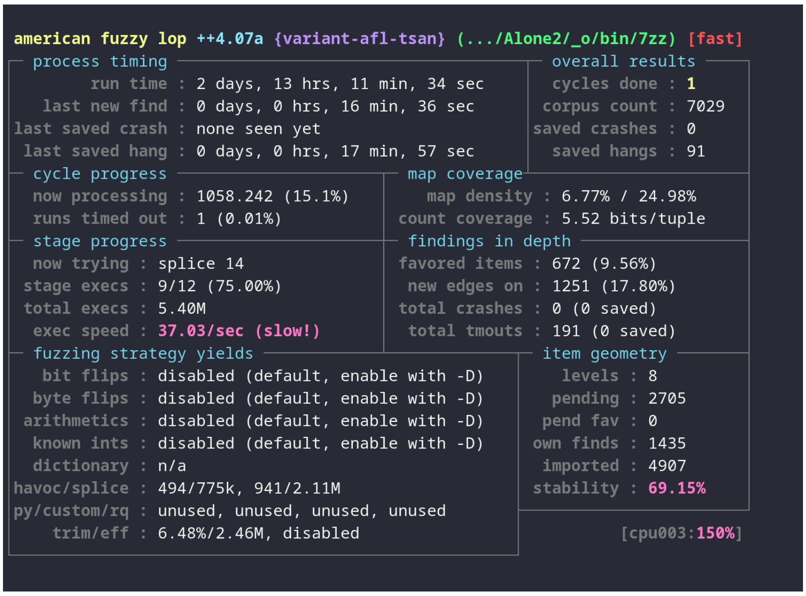
We ran the fuzzers using multiple cores for around 2.5 days. We noticed no crashes in most of the variants, with msan being the exception. However, some fuzzers encountered hangs.

Figure 1: Main AFL Fuzzer

Figure 2: ASAN Variant Fuzzer

```
american fuzzy lop ++4.07a {variant-afl-msan} (.../Alone2/_o/bin/7zz) [fast]
 process timing                                  overall results
        run time : 2 days, 13 hrs, 14 min, 31 sec   cycles done : 1
   last new find : 0 days, 0 hrs, 2 min, 53 sec    corpus count : 7427
last saved crash : 0 days, 1 hrs, 13 min, 31 sec  saved crashes : 978
 last saved hang : 0 days, 0 hrs, 36 min, 4 sec     saved hangs : 94
 cycle progress                              map coverage
  now processing : 4889*0 (65.8%)              map density : 1.05% / 5.58%
  runs timed out : 26 (0.35%)               count coverage : 5.37 bits/tuple
 stage progress                              findings in depth
  now trying : trim 8/8                     favored items : 722 (9.72%)
 stage execs : 27/90 (30.00%)                new edges on : 1327 (17.87%)
 total execs : 4.77M                        total crashes : 220k (978 saved)
  exec speed : 30.83/sec (slow!)             total tmouts : 206 (0 saved)
 fuzzing strategy yields                            item geometry
   bit flips : disabled (default, enable with -D)     levels : 5
  byte flips : disabled (default, enable with -D)    pending : 2456
 arithmetics : disabled (default, enable with -D)   pend fav : 1
  known ints : disabled (default, enable with -D)  own finds : 1268
  dictionary : n/a                                  imported : 5472
havoc/splice : 465/343k, 1270/1.46M                stability : 74.96%
py/custom/rq : unused, unused, unused, unused
    trim/eff : 6.55%/2.91M, disabled                          [cpu002:100%]
```

Figure 3: MSAN Variant Fuzzer



```
american fuzzy lop ++4.07a {variant-afl-tsan} (.../Alone2/_o/bin/7zz) [fast]
 process timing                                  overall results
        run time : 2 days, 13 hrs, 11 min, 34 sec   cycles done : 1
   last new find : 0 days, 0 hrs, 16 min, 36 sec   corpus count : 7029
last saved crash : none seen yet                  saved crashes : 0
 last saved hang : 0 days, 0 hrs, 17 min, 57 sec    saved hangs : 91
 cycle progress                              map coverage
  now processing : 1058.242 (15.1%)            map density : 6.77% / 24.98%
  runs timed out : 1 (0.01%)                count coverage : 5.52 bits/tuple
 stage progress                              findings in depth
  now trying : splice 14                    favored items : 672 (9.56%)
 stage execs : 9/12 (75.00%)                 new edges on : 1251 (17.80%)
 total execs : 5.40M                        total crashes : 0 (0 saved)
  exec speed : 37.03/sec (slow!)             total tmouts : 191 (0 saved)
 fuzzing strategy yields                            item geometry
   bit flips : disabled (default, enable with -D)     levels : 8
  byte flips : disabled (default, enable with -D)    pending : 2705
 arithmetics : disabled (default, enable with -D)   pend fav : 0
  known ints : disabled (default, enable with -D)  own finds : 1435
  dictionary : n/a                                  imported : 4907
havoc/splice : 494/775k, 941/2.11M                 stability : 69.15%
py/custom/rq : unused, unused, unused, unused
    trim/eff : 6.48%/2.46M, disabled                          [cpu003:150%]
```

Figure 4: TSAN Variant Fuzzer

We tried running an input from `in/hangs` to check where an infinite loop could occur. But, all inputs eventually terminated while taking longer than 2 seconds. Therefore, we concluded that these executions were incorrectly flagged as hangs due to relatively low timeouts. For our next fuzzing attempts, we plan to increase this and make the timout around 30 seconds to account for larger file inputs.

Analyzing msan crashes

As the msan variant was the only one that produced crashes, we compiled a msan binary with debug flags and analyzed the crash.

```
➜  fuzzing-work git:(main) ✗ ../../7zz_afl_msan_dbg/CPP/7zip/Bundles/Alone2/_o/bin/7zz e -y in/signed.jar                [15/1981]

7-Zip (z) 22.00 ZS v1.5.2 (x64) : Copyright (c) 1999-2022 Igor Pavlov : 2022-06-15
 64-bit locale=en_US.UTF-8 Threads:6

Scanning the drive for archives:
1 file, 1781 bytes (2 KiB)

Extracting archive: in/signed.jar
--
Path = in/signed.jar
Type = zip
Physical Size = 1781

  0%==3949344==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x55555575245e in NWindows::NFile::NDir::SetFileAttrib_PosixHighDetect(char const*, unsigned int) /home/ubuntu/390r-debugging-set
up/7zz_afl_msan_dbg/CPP/7zip/Bundles/Alone2/../../../Windows/FileDir.cpp:1085:12
    #1 0x555555e23629 in CArchiveExtractCallback::SetAttrib() /home/ubuntu/390r-debugging-setup/7zz_afl_msan_dbg/CPP/7zip/Bundles/Alone2
```

Figure 5: crash output from msan

As displayed above, the crash occurs due to a use of uninitialized value in *FileDir.cpp*.

```
▶ 1085   else if (S_ISLNK(st.st_mode))
  1086   {
  1087     /* for most systems: permissions for symlinks are fixed to rwxrwxrwx.
  1088        so we don't need chmod() for symlinks. */
  1089     return true;
  1090     // SetLastError(ENOSYS);
──────────────────────────────────────────[ STACK ]──────────────────────────────────────────
00:0000│ rsp 0x7fffffffcf60 ◂— 0x0
01:0008│ rdx 0x7fffffffcf68 ◂— 0x803
02:0010│     0x7fffffffcf70 ◂— 0x275c09 /* "\t\\'" */
03:0018│     0x7fffffffcf78 ◂— 0x1
04:0020│     0x7fffffffcf80 —▸ 0x3e8000081b4 ◂— 0x0
05:0028│     0x7fffffffcf88 ◂— 0x3e8
06:0030│     0x7fffffffcf90 ◂— 0x0
07:0038│     0x7fffffffcf98 ◂— 0x79 /* 'y' */
────────────────────────────────────────[ BACKTRACE ]────────────────────────────────────────
 ▶ f 0   0x55555575229c NWindows::NFile::NDir::SetFileAttrib_PosixHighDetect(char const*, unsigned int)+268
   f 1   0x555555e2362a CArchiveExtractCallback::SetAttrib()+970
   f 2   0x555555e2a609 CArchiveExtractCallback::SetOperationResult(int)+1817
   f 3   0x555555bc962f NArchive::NZip::CHandler::Extract(unsigned int const*, unsigned int, int, IArchiveExtractCallback*)+7775
   f 4   0x555555e80530 Extract(CCodecs*, CObjectVector<COpenType> const&, CRecordVector<int> const&, CObjectVector<UString>&, CObjectVector<U
rd::CCensorNode const&, CExtractOptions const&, IOpenCallbackUI*, IExtractCallbackUI*, IHashCalc*, UString&, CDecompressStat&)+20704
   f 5   0x555555e80530 Extract(CCodecs*, CObjectVector<COpenType> const&, CRecordVector<int> const&, CObjectVector<UString>&, CObjectVector<U
rd::CCensorNode const&, CExtractOptions const&, IOpenCallbackUI*, IExtractCallbackUI*, IHashCalc*, UString&, CDecompressStat&)+20704
   f 6   0x555555f3bc69 Main2(int, char**)+26233
   f 7   0x555555f45b75 main+213
─────────────────────────────────────────────────────────────────────────────────────────────
pwndbg> p st.st_mode
$1 = 33204
pwndbg> 
```

Figure 6: gdb analysis for msan crash

From previous error message, we see that msan has flagged `st.st_mode` as uninitialized. But, looking into **gdb**, this variable seems to be defined, set to 33204. This is because it was called with `lstat(path, &st)` at the beginning of the function, which initialized all fields of the struct.

From this, we can conclude that msan had incorrectly flagged this an uninitialized and this is a *false positive*. All other msan crashes refer to the same line, so we determined that **msan** is not a good fit for this project, possibly missing out on initializations.

We looked more into clang documentation, which confirmed this hypothesis:

> it may introduce false positives and therefore should be used with care

Static Analysis

Codeql

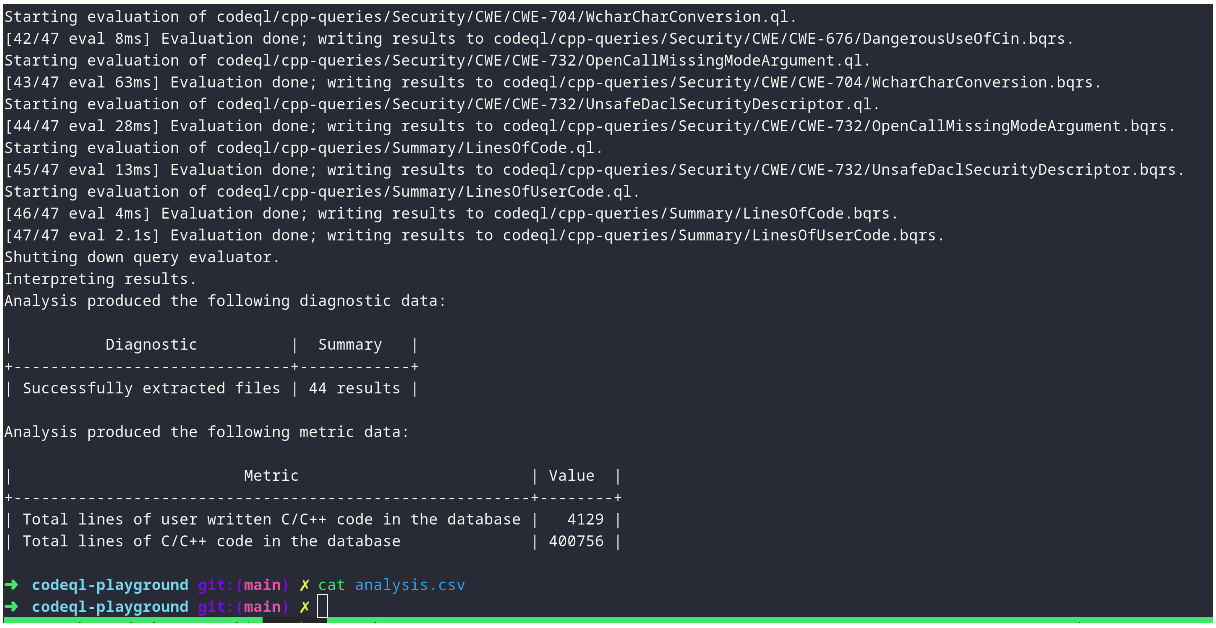
To analyze the code for common C/C++ bugs, we used **codeql** to scan the source code.

We first created a analysis database by providing the `make` instructions to codeql.

```
codeql database create ../../../../../codeql-playground/analysis-db.
    codeql -l cpp -c "make -B -f makefile.gcc" --overwrite
```

Then, we download *cpp-queries* and tested the produced database against it.

```
codeql pack download codeql/cpp-queries
codeql database analyze analysis-db.codeql --format CSV --output
    analysis.csv
```



```
Starting evaluation of codeql/cpp-queries/Security/CWE/CWE-704/WcharCharConversion.ql.
[42/47 eval 8ms] Evaluation done; writing results to codeql/cpp-queries/Security/CWE/CWE-676/DangerousUseOfCin.bqrs.
Starting evaluation of codeql/cpp-queries/Security/CWE/CWE-732/OpenCallMissingModeArgument.ql.
[43/47 eval 63ms] Evaluation done; writing results to codeql/cpp-queries/Security/CWE/CWE-704/WcharCharConversion.bqrs.
Starting evaluation of codeql/cpp-queries/Security/CWE/CWE-732/UnsafeDaclSecurityDescriptor.ql.
[44/47 eval 28ms] Evaluation done; writing results to codeql/cpp-queries/Security/CWE/CWE-732/OpenCallMissingModeArgument.bqrs.
Starting evaluation of codeql/cpp-queries/Summary/LinesOfCode.ql.
[45/47 eval 13ms] Evaluation done; writing results to codeql/cpp-queries/Security/CWE/CWE-732/UnsafeDaclSecurityDescriptor.bqrs.
Starting evaluation of codeql/cpp-queries/Summary/LinesOfUserCode.ql.
[46/47 eval 4ms] Evaluation done; writing results to codeql/cpp-queries/Summary/LinesOfCode.bqrs.
[47/47 eval 2.1s] Evaluation done; writing results to codeql/cpp-queries/Summary/LinesOfUserCode.bqrs.
Shutting down query evaluator.
Interpreting results.
Analysis produced the following diagnostic data:

|         Diagnostic          |  Summary   |
+-----------------------------+------------+
| Successfully extracted files | 44 results |

Analysis produced the following metric data:

|                       Metric                          | Value  |
+-------------------------------------------------------+--------+
| Total lines of user written C/C++ code in the database |   4129 |
| Total lines of C/C++ code in the database              | 400756 |

➜ codeql-playground git:(main) ✗ cat analysis.csv
➜ codeql-playground git:(main) ✗
```

Figure 7: Codesql statistics

But, this did not output any glaring errors, executing without any warnings or useful metrics. We concluded that codeql only utilizes simple checks which would have already been accounted for in the source code.

CPPCheck

We ran the codebase through the static analysis tool **cppcheck**, which tagged 1569 warnings and errors. One of the common errors flagged by cppcheck was *shiftTooManyBits*

390r-debugging-setup\p7zip\CPP\7zip\Archive\7z\7zIn.cpp

261	shiftTooManyBits	758	error	Shifting 32-bit value by 32 bits is undefined behaviour
1546	shiftTooManyBits	758	error	Shifting 32-bit value by 32 bits is undefined behaviour
1547	shiftTooManyBits	758	error	Shifting 32-bit value by 32 bits is undefined behaviour
1598	shiftTooManyBits	758	error	Shifting 32-bit value by 62 bits is undefined behaviour

Figure 8: Occurences of *shiftTooManyBits*

Unfortunately, when looking at the actual source code, almost all of these errors come from an innocuous macro:

```
#define GetUi64(p) (GetUi32(p) | ((UInt64)GetUi32(((const Byte *)(p)) + 4) << 32))
```

Figure 9: First macro

```
#define GetUi32(p) ( \
             ((const Byte *)(p))[0]        | \
    ((UInt32)((const Byte *)(p))[1] <<  8) | \
    ((UInt32)((const Byte *)(p))[2] << 16) | \
    ((UInt32)((const Byte *)(p))[3] << 24))
```

Figure 10: Second macro

The rest, on closer inspection, are also falsely flagged as errors, such as this one:

2594	shiftTooManyBits	758	error	Shifting 32-bit value by 63 bits is undefined behaviour

Figure 11: Another occurence of *shiftTooManyBits*

```
if (node.FileSize >= ((UInt64)1 << 63))
    return S_FALSE;
```

Figure 12: Looking into source code

A more promising error seems to be a possible null pointer exception:

390r-debugging-setup\p7zip\CPP\7zip\Archive\Zip\ZipHandlerOut.cpp

41	nullPointerRedundantCheck	476	warning	Either the condition 'password' is redundant or there is possible null pointer dereference: s++.
41	nullPointerArithmeticRedundantCheck	682	warning	Either the condition 'password' is redundant or there is pointer arithmetic with NULL pointer.

Figure 13: Null Pointer occurences

```
37 static bool IsSimpleAsciiString(const wchar_t *s)
38 {
39   for (;;)
40   {
41     wchar_t c = *s++;
42     if (c == 0)
43       return true;
44     if (c < 0x20 || c > 0x7F)
45       return false;
46   }
47 }
```

Figure 14: First null Pointer dereference

This function is only called once, in the same file at line 415:

```cpp
CMyComPtr<ICryptoGetTextPassword2> getTextPassword;
{
  CMyComPtr<IArchiveUpdateCallback> udateCallBack2(callback);
  udateCallBack2.QueryInterface(IID_ICryptoGetTextPassword2, &getTextPassword);
}
CCompressionMethodMode options;
(CBaseProps &)options = _props;
options._dataSizeReduce = largestSize;
options._dataSizeReduceDefined = largestSizeDefined;

options.PasswordIsDefined = false;
options.Password.Wipe_and_Empty();
if (getTextPassword)
{
  CMyComBSTR_Wipe password;
  Int32 passwordIsDefined;
  RINOK(getTextPassword->CryptoGetTextPassword2(&passwordIsDefined, &password));
  options.PasswordIsDefined = IntToBool(passwordIsDefined);
  if (options.PasswordIsDefined)
  {
    if (!m_ForceAesMode)
      options.IsAesMode = thereAreAesUpdates;

    if (!IsSimpleAsciiString(password))
      return E_INVALIDARG;
```

Figure 15: Second null Pointer dereference

It looks like `password` gets populated in *CryptoGetTexPassword2*, looking at that function, and the subsequent call to *StringToBstr*, it unfortunately looks like the nullpointer is properly checked for.

```cpp
STDMETHODIMP CUpdateCallback100Imp::CryptoGetTextPassword2(Int32 *passwordIsDefined, BSTR *password)
{
  *password = NULL;
  *passwordIsDefined = BoolToInt(PasswordIsDefined);
  if (!PasswordIsDefined)
    return S_OK;
  return StringToBstr(Password, password);
}
```

Figure 16: Null pointer check in *CryptoGetTexPassword2*

```cpp
inline HRESULT StringToBstr(LPCOLESTR src, BSTR *bstr)
{
  *bstr = ::SysAllocString(src);
  return (*bstr) ? S_OK : E_OUTOFMEMORY;
}
```

Figure 17: Null pointer check in *StringToBstr*

Further Thoughts

We analyzed our target with **cppcheck** and **codeQL**. These utilities have obvious limitations, but as an aside we thought it interesting to explore any existing work documenting the effectiveness of each. A recent 2022 study titled "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection", found that in general, cppcheck was ineffective at identifying vulnerabilities in their tested scenarios when run alone (Lipp et al., 2022). Further, among the tested scenarios, **codeQL** was second to the industry utility **CommSCA**; however, even in the best case scenario, 30% of the 192 vulnerabilities tested were not detected.

Ultimately, it seems that static analysis with standard utilities of this particular target may not prove particularly fruitful, since both CPPCheck and Codeql failed to find anything particularly interesting. As such, our coming work when it comes to static analysis sshould involve getting LLVM bitcode at some point in compilation, and then writing queries for vulnerable functions such as **malloc()**.

Next Steps

- Fuzzing with higher timeout to increase coverage and avoid false positives
- Fuzzing different features of the target binary
- Investigating how *p7zip* is fuzzed and tested in OSS-Fuzz
- Figure out how to inject LLVM pass and emit bitcode, then write queries