InfoZip UnZip 6.00 / 6.1c22 Buffer Overflow

February 07, 2018

Exploit Collector

InfoZip UnZip versions 6.00 and below and 6.1c22 and below suffer from multiple buffer overflow vulnerabilities.

MD5 | `bdf125c9b1ccf7ea7ce8e8e8062e3d85`

Download

```
              product: InfoZip UnZip
 vulnerable version: UnZip <= 6.00 / UnZip <= 6.1c22
      fixed version: 6.10c23
         CVE number: CVE-2018-1000031,CVE-2018-1000032,CVE-2018-1000033
                     CVE-2018-1000034,CVE-2018-1000035
             impact: high
           homepage: http://www.info-zip.org/UnZip.html
              found: 2017-11-03
                 by: R. Freingruber (Office Vienna)
                     SEC Consult Vulnerability Lab

                     An integrated part of SEC Consult
                     Bangkok - Berlin - Linz - Luxembourg - Montreal - Moscow
                     Kuala Lumpur - Singapore - Vienna (HQ) - Vilnius - Zurich

                     https://www.sec-consult.com

=======================================================================

Vendor description:
-------------------
"UnZip is an extraction utility for archives compressed in .zip format (also
called "zipfiles"). Although highly compatible both with PKWARE's PKZIP and
PKUNZIP utilities for MS-DOS and with Info-ZIP's own Zip program, our
primary objectives have been portability and non-MSDOS functionality.
UnZip will list, test, or extract files from a .zip archive, commonly found
on MS-DOS systems. The default behavior (with no options) is to extract into
the current directory (and subdirectories below it) all files from the
specified zipfile."

Source: http://www.info-zip.org/UnZip.html

InfoZip's UnZip is used as default utility for uncompressing ZIP archives
on nearly all *nix systems. It gets shipped with many commerical products on
Windows to provide (un)compressing functionality as well.


Business recommendation:
------------------------
InfoZip Unzip should be updated to the latest available version.


Vulnerability overview/description:
-----------------------------------
```

```
For newer builds the risk for this vulnerability is partially mitigated
because modern compilers automatically replace unsafe functions with length
checking variants of the same function (for example sprintf gets replaced
by sprintf_chk). This is done by the compiler at locations were the length
of the destination buffer can be calculated.

Nevertheless, it must be mentioned that UnZip is used on many systems
including older systems or on exotic architectures on which this protection
is not in place. Moreover, pre-compiled binaries which can be found on the
internet lack the protection because the last major release of InfoZip's
UnZip was in 2009 and compilers didn't enable this protection per default at
that time. The required compiler flags are also not set in the Makefile of
UnZip. Compiled applications are therefore only protected if the used compiler
has this protection enabled per default which is only the case with modern
compilers.

To trigger this vulnerability (and the following) it's enough to uncompress
a manipulated ZIP archive. Any of the following invocations can be used to
trigger and abuse the vulnerabilities:

>unzip malicious.zip
>unzip -p malicious.zip
>unzip -t malicious.zip

2) Heap-based out-of-bounds write (CVE-2018-1000031)

This vulnerability only affects UnZip 6.1c22 (next beta version of UnZip).
InfoZip's UnZip suffers from a heap-based out-of-bounds write if the
archive filename does not contain a .zip suffix.

3) Heap/BSS-based buffer overflow (Bypass of CVE-2015-1315) (CVE-2018-1000032)

This vulnerability only affects UnZip 6.1c22 (next beta version of UnZip).
InfoZip's UnZip suffers from a heap/BSS-based buffer-overflow which
can be used to write null-bytes out-of-bound when converting
attacker-controlled strings to the local charset.

4) Heap out-of-bounds access in ef_scan_for_stream (CVE-2018-1000033)

This vulnerability only affects UnZip 6.1c22 (next beta version of UnZip).
InfoZip's UnZip suffers from a heap out-of-bounds access
vulnerability.
```

```
implementation. Various crash dumps have been supplied to the vendor
but no furthed analysis has been performed.


Proof of concept:
-----------------
1) Heap-based buffer overflow in password protected ZIP archives (CVE-2018-1000035)

Unzipping a malicious archive results in the following output:
(On Ubuntu 16.04 with UnZip 6.0 which was installed via aptitude install unzip)

*** buffer overflow detected ***: unzip terminated
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x*****)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(__fortify_fail+0x**)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(+0x*****)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(+0x*****)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(_IO_default_xsputn+0x**)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(_IO_vfprintf+0x**)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(__vsprintf_chk+0x**)[0x************]
/lib/x86_64-linux-gnu/libc.so.6(__sprintf_chk+0x**)[0x************]
unzip[0x40c02b]
unzip[0x4049ac]
unzip[0x40762c]
unzip[0x409b60]
unzip[0x411175]
unzip[0x411bdf]
unzip[0x404191]

Function names can be mapped to the backtrace by compiling the application
with debug symbols:

(gdb) backtrace
#0  0x000000000040c706 in UzpPassword ()
#1  0x00000000004043ce in decrypt ()
#2  0x000000000040731c in extract_or_test_entrylist ()
#3  0x00000000004094af in extract_or_test_files ()
#4  0x00000000004149a5 in do_seekable ()
#5  0x000000000041540f in process_zipfiles ()
#6  0x0000000000403921 in unzip ()

The vulnerability resides inside the UzpPassword function in the following
code snippet (file ./fileio.c):
```

```
[1595]  }

The allocation at line 1591 allocates a fixed size buffer and then writes into
it at line 1592. It writes the following format string (PasswPrompt) into
the buffer: "[%s] %s password: "

This string has a length of 15 including the null-termination which explains
the +15 in the allocation. The developer allocated 2*FILENAMESIZ which
corresponds to 2 * PATH_MAX for the two format strings (zfn and efn).
zfn is the archive filename and can therefore not exceed PATH_MAX.
efn is the current processed filename inside the ZIP archive which should
typically be smaller than PATH_MAX for normal files. However, since an
attacker can manipulate the archive file the name can arbitrarily be chosen
which leads to a heap-based buffer overflow.

As already mentioned, modern compilers replace unsafe functions with
safe alternatives as a defense in depth mechanism.
This feature is called BOSC (Built-in object size checking) and is part
of the FORTIFY_SOURCE=2 protection.
The following link shows the source code (and vulnerability) inside
the Ubuntu package:
http://bazaar.launchpad.net/~ubuntu-branches/ubuntu/trusty/unzip/trusty-updates/view/head:/fileio.c#L1593

By checking the installed compiled binary the following code can be seen:
(UnZip 6.0 from Ubuntu 16.04)

0x40bfc6:    mov    edi,0x200f
0x40bfcb:    mov    r13,r8
0x40bfce:    mov    QWORD PTR [rsp+0x8],r9
0x40bfd3:    call   0x401d30 <malloc@plt>
...
0x40c01a:    mov    edx,0x200f
0x40c01f:    mov    esi,0x1
0x40c024:    xor    eax,eax
0x40c026:    call   0x401f40 <__sprintf_chk@plt>

The code allocates 0x200f (=4096*2 + 15) bytes but the unsafe sprintf
function was replaced with the length-checking sprintf_chk() function
which receives as argument the size of the buffer (0x200f at address
0x40c01a). The risk is therefore mitigated on Ubuntu (and other modern
operating systems), at least with the currently used compiler default flags.

However, many pre-compiled UnZip binaries can be found on the internet
```

```
https://oss.oracle.com/el4/unzip/unzip.tar
http://www.willus.com/archive/zip64/


2) Heap-based out-of-bounds write (CVE-2018-1000031)

When uncompressing ZIP archives the following code gets executed
(file fileio.c:345 function set_zipfn_sgmnt_name() in UnZip 6.1c22):

#define SGMNT_NAME_BOOST 8
...
  if (G.zipfn_sgmnt == NULL)
  {
[1]     G.zipfn_sgmnt_size = (int)strlen(G.zipfn)+ SGMNT_NAME_BOOST;
    if ((G.zipfn_sgmnt = izu_malloc(G.zipfn_sgmnt_size)) == NULL)
  ...
[2]   zfstrcpy(G.zipfn_sgmnt, G.zipfn);
  /* Expect to find ".zXX" at the end of the segment file name. */
[3]   sufx_len = IZ_MAX( 0, ((int)strlen(G.zipfn_sgmnt)- 4));
[4]   suffix = G.zipfn_sgmnt+ sufx_len;
  ...
  else  // No .zip extension
  {
[5]     zfstrcpy( (suffix+ sufx_len), ZSUFX);
[6]     suffix += sufx_len+ 2;
  }
  /* Insert the next segment number into the file name (G.zipfn_sgmnt). */
[7]   sprintf(suffix, "%02d", (sgmnt_nr+ 1));

G.zipfn is the filename / path of the archive file. Line [1] allocates space
for this name plus 8 (SGMNT_NAME_BOOST). Line [2] copies the name.
[3] and [4] calculate the end address minus 4 which should point to a suffix
if one is present. After [4] the variable suffix already points to this
address. However, line [5] adds sufx_len again to suffix, the write target
is therefore the base address + 2*(allocation_length - 4) but the buffer
can only hold allocation_len bytes.
Line [7] is another out-of-bounds write because of line [6].

Memory trace of the crash:
#1 0xf7af3c2f in memcpy (/usr/lib/i386-linux-gnu/libasan.so.2+0x8ac2f)
#2 0x80969be in set_zipfn_sgmnt_name unzip610c22/fileio.c:424
#3 0x808eb82 in find_local_header unzip610c22/extract.c:4469
#4 0x808eb82 in extract_or_test_entrylist unzip610c22/extract.c:4745
```

```
#10 0x804a5b6 in main unzip610c22/unzip.c:1280
#11 0xf78cb636 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18636)

Please note that this vulnerability must not lead to a crash. If the
overwritten memory is not used, the program works as expected.


3) Heap/BSS-based buffer overflow (Bypass of CVE-2015-1315) (CVE-2018-1000032)

During parsing ZIP archives the function charset_to_intern() can be called
(unix/Unix.c:2427) with the "string" argument pointing to attacker
controlled data. This function converts the string in-place to another
charset (string is an INOUT argument).

The following code performs this task in the function:
[1]    slen = strlen(string);
[2]    s = string;
[3]    dlen = buflen = 2 * slen;
[4]    d = buf = izu_malloc(buflen + 1);
    if (d)
    {
[5]      memset( buf, 0, buflen);

      /* 2015-02-12 William Robinet, SMS.  CVE-2015-1315.
       * Added FILNAMSIZ check to avoid buffer overflow. Better would
       * be to pass in an actual destination buffer size.
       */
[6]      if ((iconv(cd, &s, &slen, &d, &dlen) != (size_t)-1) &&
[7]       (strlen(buf) < FILNAMSIZ))
      {
[8]        strncpy(string, buf, buflen);
      }
      izu_free(buf);
    }

The input string pointer is stored in the variable "string" and "s" (see [2]).
Line [6] tries to convert the input ("s") via iconv() to another charset.
The destination is "d" / "buf" (see line [4]).
This destination buffer has a size of two times the input length plus one ([3]).

The first problem can be found in line [5] which just initializes "buflen" bytes
and not "buflen+1" bytes. Consider the input string is empty, therefore slen=0.
This will allocate 1 byte at line [4] because of the +1. However, [5] calls
```

```
The second problem is harder to identify. The function do_string() is used to
parse strings from ZIP archives. If the option DS_FN gets passed, the string
is written into the filename[] array from the global variable G.
The code at extract.c:5584 (in the function extract_or_test_files()) calls
for example this function with this option:

do_string(__G__ G.crec.filename_length, DS_FN)) != PK_COOL)

Inside do_string() the following code can be found (fileio.c:3225):

Ext_ASCII_TO_Native(G.filename, G.pInfo->hostnum, G.pInfo->hostver,
  G.pInfo->HasUxAtt, (option == DS_FN_L));

The "Ext_ASCII_TO_Native" is a define which redirects to charset_to_intern().
The first argument (G.filename in this case) is passed to this function
and can be accessed with the "string" argument in the above code.
At line [6] iconv() is used to convert the input string from one charset
(e.g.: CP850) to another (e.g.: UTF-8). Therefore "buf" contains the
converted string after this call. With line [8] this converted string
should be copied over the original location from the argument
(G.filename in our case).

The strncpy at [8] limits the number of written characters to buflen.
Because of [3] buflen is two times the input length and therefore a buffer
overflow can happen (the real size of the input buffer is not passed to the
function). This vulnerability was CVE-2015-1315 and an additional check was
added to prevent this buffer overflow. The additional check at [7] checks
the length of the converted string with this line:
[7]       (strlen(buf) < FILNAMSIZ))

Only if this check is passed the code at [8] gets executed:

[8]         strncpy(string, buf, buflen);

This should logically limit the number of bytes which can be written to
be smaller than FILNAMSIZ (even if the wrong, higher number, is passed
to strncpy).

For example:
G.filename is defined in globals.h:372 (inside the Uz_Globs struct):

char     filename[FILNAMSIZ];
```

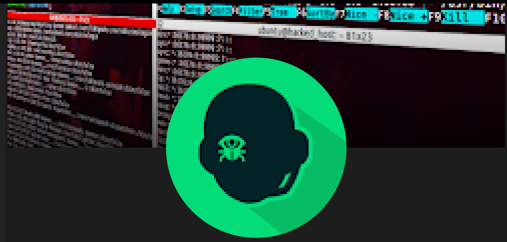
```
[6]      if ((iconv(cd, &s, &slen, &d, &dlen) != (size_t)-1) &&
[7]       (strlen(buf) < FILNAMSIZ))
      {
[8]        strncpy(string, buf, buflen);
      }

Let's assume that our input string had a length of 2940. Because of
[3] buflen will be 5880 (2*2940). That means if [8] is reached
a strncpy with a limit of 5880 gets executed, however, the destination
buffer only has a size of 4096 bytes (G.filename).
The check at [7] should protect against this because if strlen(buf) (the
source from strncpy) is bigger or equal than FILNAMSIZ (4096), the
strncpy does not get executed. And since strncpy just copies until the
first null-byte, it should just be possible to copy at maximum strlen(buf)
bytes in this strncpy.

This assumption is wrong though.
Strncpy() always writes n bytes - in the above case it will always write
5880 bytes and therefore a buffer overflow will always occur.
This behavior can be found in the manpage of strncpy:
"If the length of src is less than n, strncpy() writes additional null
bytes to dest to ensure that a total of n bytes are written."

The strncpy can therefore be used to write null-bytes out-of-bound
in the BSS or heap segment. Since the input string length is under
attacker control, the write length can be manipulated. That means
that an attacker can perform a partial overwrite to exploit this
vulnerability. For example, the attacker can overwrite data in the
Uz_Globs struct after G.filename with null-bytes. One attack target
can be heap addresses. They can be partially overwritten (lower
bytes) to change the heap address to point to an attacker
controlled heap chunk to get control over the data and therefore
also over the execution.


4) Heap out-of-bounds access in ef_scan_for_stream (CVE-2018-1000033)

The first two arguments to the function ef_scan_for_stream()
(extract.c:1167) are: ef_ptr and ef_len.
This function is for example called at: extract.c:4795

sts = ef_scan_for_stream( G.extra_field,
  (long)G.lrec.extra_field_length,
```

```
The second argument (ef_len) stores the length / size of the first
argument (ef_ptr) and access checks must be performed to ensure
that no out-of-bounds access occurs.

Code line extract.c:1233 can access data out-of-bounds because length
checks are missing:

bitmap = *(ef_ptr+ (data_byte++));

Debugger output:

Program received signal SIGSEGV, Segmentation fault.
ef_scan_for_stream (...) at extract.c:1233
1233            bitmap = *(ef_ptr+ (data_byte++));
(gdb) print /x ef_ptr
$10 = 0x7ffff7ed5f3b
(gdb) print /x data_byte
$11 = 0xc6
(gdb) print /x ef_len
$12 = 0xc5



5) Multiple vulnerabilities in the LZMA compression algorithm (CVE-2018-1000034)

Invalid access attempts can occur at:
szip/LzmaDec.c:275 - IF_BIT_0(probLen)
szip/LzmaDec.c:242 - IF_BIT_0(prob)
szip/LzmaDec.c:217 - IF_BIT_0(prob)
szip/LzmaDec.c:299 - TREE_6_DECODE(prob, distance);
szip/LzmaDec.c:189 - GET_BIT2(probLit, symbol, offs &= ~bit, offs &= bit)
szip/LzmaDec.c:264 - IF_BIT_0(probLen)
szip/LzmaDec.c:201 - IF_BIT_0(prob)
szip/LzmaDec.c:213 - IF_BIT_0(prob)
szip/LzmaDec.c:290 - TREE_DECODE(probLen, limit, len);
szip/LzmaDec.c:233 - IF_BIT_0(prob)

No further analysis has been performed on the LZMA compression code.
The vendor will remove this code entirely in future releases.


Vulnerable / tested versions:
-----------------------------
```

```
Vendor contact timeline:
------------------------
2017-11-03: Vulnerability 1 identified, further internal analysis
2017-11-06: Attempt to contact the developers via bug report page
2017-11-10: Initial contact to the developer via sms@antinode.info
2017-11-10: Information from the main developer: A new beta version (6.1c22),
            which will be released soon, incorporates some security features.
            A link to the new beta version was provided.
2017-11-12: Sending encrypted advisory to sms@antinode.info
            Informed developer of the latest possible release date (2017-12-30).
2017-11-13: Developer confirms the vulnerability and notes that
            it should be easy to fix. The developer asks for a notification
            if vulnerabilities are found in version 6.1c22.
2017-11-21: Vulnerability 2-5 in UnZip 6.1c22 identified,
            the updated encrypted advisory with crash files was sent to
            the developer.
2017-11-23: Developer confirmed the e-mail containing the updated advisory.
2017-12-06: Asking the developer when an update will be available and
            to coordinate the release of the advisory together.
2017-12-11: E-mail from the developer: All vulnerabilities (except LZMA
            vulnerabilities) are fixed in version 6.1c23. A link to the new
            version was provided. The LZMA code / feature will likely be disabled
            until a better solution is available.
2017-12-13: Asking the developer for a coordinated release of the advisory.
2018-01-04: Informing the developer about the changed release date because
            of the holidays. Distribution mailing lists will be informed on
            2018-01-17, the advisory will be released about one week after that.
            Asking the developer for an InfoZip version with LZMA disabled.
2018-01-10: Informing the developer again that the current solution is to
            upgrade to version 6.10c23 which still contains the LZMA
            vulnerabilities and if a version without LZMA is available.
2018-01-17: Informing distros@vs.openwall.org about the upcoming advisory.
2018-02-01: Received CVE numbers.
2018-02-07: Publication of the advisory



Solution:
---------
Update to version 6.10c23: http://antinode.info/ftp/info-zip/unzip610c23.zip
Please note that the LZMA vulnerabilities are not yet fixed in this version.
```

```
Advisory URL:
-------------
https://www.sec-consult.com/en/vulnerability-lab/advisories/index.html


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SEC Consult Vulnerability Lab

SEC Consult
Bangkok - Berlin - Linz - Luxembourg - Montreal - Moscow
Kuala Lumpur - Singapore - Vienna (HQ) - Vilnius - Zurich

About SEC Consult Vulnerability Lab
The SEC Consult Vulnerability Lab is an integrated part of SEC Consult. It
ensures the continued knowledge gain of SEC Consult in the field of network
and application security to stay ahead of the attacker. The SEC Consult
Vulnerability Lab supports high-quality penetration testing and the evaluation
of new offensive and defensive technologies for our customers. Hence our
customers obtain the most current information about vulnerabilities and valid
recommendation about the risk profile of new technologies.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Interested to work with the experts of SEC Consult?
Send us your application https://www.sec-consult.com/en/career/index.html

Interested in improving your cyber security with the experts of SEC Consult?
Contact our local offices https://www.sec-consult.com/en/contact/index.html
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Mail: research at sec-consult dot com
Web: https://www.sec-consult.com
Blog: http://blog.sec-consult.com
Twitter: https://twitter.com/sec_consult

EOF R. Freingruber @2018
```

Related Posts



KitPloit

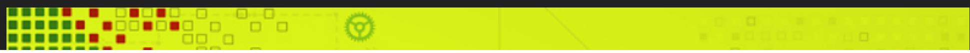
google.com/+KitploitWeb

G+ Follow

Popular Posts

Linux/x86 Read /etc/passwd Shellcode

62 bytes small Linux/x86 read /etc/passwd shellcode.



Exploit Collector

Microsoft Internet Explorer VBScript Engine CVE-2018-8174 Arbitrary Code Execution Vulnerability

Microsoft Internet Explorer is prone to an unspecified arbitrary code-execution vulnerability.

Attackers can exploit this vulnerability to execute arbitrary code in the …

WhatsApp 2.18.31 iOS Memory Corruption

WhatsApp version 2.18.31 on iOS suffers from a remote memory corruption vulnerability.

Archive