1 Introduction

2 Determine library version

3 Conclusion
Section 1

Introduction
Introduction

Hackers look for vulnerabilities in programs;
Sometimes, finding one allow them to make an exploit;
Developers try to fix vulnerabilities, publishing a patch;
Users install the patch to hopefully fix the vulnerability on their version of the program.
What could go wrong?

The vulnerability is not necessarily patched and the old version may remain;

Very dangerous!
Which libraries are used by a program?

What is the version of OpenSSL?

```
ls -1a
.
libAAX_SDK.so
libintaudio_i.so
libintaudio_j2.so
libintaudio_j.so
libintaudio_k.so
libintaudio_l.so
libprhlpr.so
```

```
ldd libintaudio_k.so
linux-gate.so.1 => (0xf77ee000)
libprhlpr.so => not found
libdl.so => /usr/lib32/libdl.so (0xf76c6000)
liblog.so => not found
libz.so => not found
libmedia.so => not found
libutils.so => not found
libandroid_runtime.so => not found
libbinder.so => not found
libnativehelper.so => not found
libcrypto.so => not found
libstagedright.so => not found
libssl.so => not found
libexpat.so => not found
libOpenSLES.so => not found
libstdc++.so => not found
lib.so => not found
libm.so => /usr/lib32/libm.so (0xf766d000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf74b7000)
/lib/ld-linux.so.2 (0x56589000)
```
Introduction
Determine library version
Conclusion

Use your hands

Strings

- Strings command gives binary infos.

- It does not work for zlib
  - We do not know any other methods. But pros do.
  - Is there any pro here?
Section 2

Determine library version
Overview - How to check if we applied a patch?

What do we have?
- An unknown .so library, $L$;
- A set of known libraries sources, $S$.

What are we doing?
For all library $s$ in $S$, we check if $L$ is of the same version as $s$. 
Lazy start

- Compile libraries in $S, c_1, \ldots, c_n$;
- Compare bit by bit $L$ and $c_1, \ldots, c_n$.

Issues

$L$ is compiled with a different compiler and different compiling options.
Be smarter

Ideas

- Link compatibility;
- Malware signature;
- Traces.
Link dynamically the library
Link dynamically the library

![Diagram showing the process of linking a library:
2. Include a header file.
3. Link the compiled C file with the library.
4. Compile the main.c file.
5. Include the library.
6. The resulting binary is linked with the library.]
What do we need?

**Scripts**

- Generate .so from libraries: OpenSSL, Zlib, libXml, ...
- Generate different versions for each library;
- Generate test program calling library functions;
- Link program with unknown library.
Turnkey programs

Build the program

- Checking major version of a library with link-testing;
  - Semi-auto-compiling libraries;
  - Semi-auto-extracting headers;
  - Auto-generation of a c program to link-test libraries.

Tested with OpenSSL 1.0.2h

- OpenSSL 1.0.2j - cannot distinguish;
- OpenSSL 1.0.1e - can distinguish.

Tested with Zlib 1.2.8

- Zlib 1.2.7 - cannot distinguish;
- Zlib 1.1.3 - can distinguish.
Malware signature

Control flow graph

- Representing binary with a CFG;
- Using reduced CFG (Gorille);
- Should work for major versions;
- Not adapted to distinguish minor versions.

Open questions

- Distinguish minor patch from optimisation?
- How security patch more impacts binaries?
Analyse traces

- Analyse program behaviour;
- Analyse functions.

Fuzzing

- Does not work with references;
- Smart fuzzing would be to use library tests.

DSE

Should work to distinguish functions
Section 3

Conclusion
Conclusion

**Issues**
- Tools are too specific to some cases, not our cases.

**Results**
- Distinguish major versions;
- Some ideas to improve results.
It brought us..

Feelings
- Inner peace;
- Intimacy;
- We lost faith in formal methods, for binary analysis at least.

Technical
- We got back our Perl, that is priceless;
- Binaries are complex but fun.