Goal

To let you understand

- why build problems are going to be important for your static analysis research

- how you can overcome them to some extent
Contents

- Build Problem, Inevitable
- Techniques for Following Build Processes
- Possible Improvements
Build Problem, Inevitable
for static source code analysis
Imagine you have the ultimate static analyzer: sound & complete, 100% recall & precision
Will you be able to use its full power without knowing what to analyze?
Almost every program is built by compiling multiple source codes.
Almost every program is built by compiling multiple source codes.
And linking objects and libraries which results from multiple set of build processes
So, it is crucial to using your static analyzer, to know which and how source codes are built.
So, it is crucial to using your static analyzer, to know which and how source codes are built.
Countless Build Tools & Systems

- GNU Make, BSD Make, imake, nmake, ...
- configure scripts
- autoconf, automake
- CMake, qmake
- Scons
- .dsp, .dsw, .sln
- shell scripts, batch files
- Ant, Maven
- Eclipse .project, .cproject
- Debian, SRPM, Gentoo, Ports, Pkgsrc, MacPorts, ...
- ...

Plan A: Analyze Build Rules Statically?

- Too complex
- Too many tools, systems, languages
- Too many blackboxes (unknown tools)
- Too many dynamic factors (env. variables)
Build tools are fancy ways to invoke compilers, linkers, and many other tools e.g. lex, yacc, cc, ld, strip, ar, ranlib, ...

We are interested in which and how source codes are built (*.c, -I, -D arguments to gcc)
Plan B: Dynamic Analysis

If we can’t do it statically, let’s try a dynamic analysis of builds!
Techniques for Following Build Processes in POSIX environment
Replacing Executables

Replace executables of tools of interest

/usr/bin/gcc

/usr/bin/ld

...
Replacing Executables

Replace executables of tools of interest

- /usr/bin/gcc
- /usr/bin/ld
- ...

Pros

- Can intercept any tool invocation!
Replacing Executables

Replace executables of tools of interest

- /usr/bin/gcc
- /usr/bin/ld
- ...

Pros
- Can intercept any tool invocation!

Cons
- Need to mangle system (requires root permission!)
PATH manipulation

Modify PATH env. variable

/usr/bin/gcc
/tmp/bin/gcc

PATH=/tmp/bin:
$PATH
PATH manipulation

Modify PATH env. variable

- /usr/bin/gcc
- /tmp/bin/gcc
- PATH=/tmp/bin:
  $PATH

Pros

- No need of root permission, change to the system
PATH manipulation

Modify PATH env. variable

- /usr/bin/gcc
- /tmp/bin/gcc
- PATH=/tmp/bin:$PATH

Pros

- No need of root permission, change to the system

Cons

- Can’t see absolute path name invocations
amake of Airac

- Prepares fake/hook executables
- Manipulates PATH
- Collects compiler, linker invocations
- Saves preprocessed source codes separately so anyone can run analyzer on them later
Possible Improvements
Intercepting exec*(2) calls

Any program execution must go thru an exec system call

libc.so is the standard C library that contains them: execl, execle, execlp, execv, execvp, ...

LD_PRELOAD variable allows one to preload shared libraries

Drawback: no use for statically linked executables
observe

- Is an open-source tool (written by me :)
  - https://github.com/netj/observe/
- Allows you to specify rules for hooking processes
- Supports drivers: libc, PATH, PATH (WIN32)
observe Rule Example

hook gcc ls:

success=echo "$@" >>args.log

failure=echo "$@" >>failed.log
observe Demo
Process monitoring from Windows

- By hooking CreateProcess
- PsSetCreateProcessNotifyRoutine
- WMI: Win32_ProcessStartTrace
Recap

- Knowing which and how source codes are built is crucial to static source code analysis.
- Build processes are complex.
- But you can, at least, extract a fair amount of information by following them.
- And there’s a lot more you can do to improve.