Vulnerability Research

Reverse Engineering – Vulnerability Research
Sergi Àlvarez aka pancake

Vulnerability Research

Vulnerability research (3h)

Theory

- What is a bug? How can we know if it's exploitable? What's the risk? How to fix them if we don't have the source?
- Fuzzing and bruteforcing.
- Good and bad development practices
- Identifying dangerous libc functions
- Finding changes on the new binary after an upgrade
- Signatures and static linking, finding known bugs on statically linked libraries

Practice

- Find vulnerabilities/weakness/problems/bad-practices in some binaries
- Do fuzzing with afl, zuff or radamsa (nightmare?)
- Understanding crash logs, removing dupes and identifying

Bugs…

What’s a bug?

- Programming, compiler or hardware mistake that can result in execution misbehaviour
- The origin of this name comes from the first unexpected behaviour that was happening in computer science and it was caused by a real life bug burned between the computer wires and causing a short circuit

What’s a Vulnerability?

- Programming, compiler or hardware mistake that allows an attacker to exploit it for its own benefit
- The benefit of exploiting a vulnerability can be either execute code, bypass security protections by accessing forbidden resources or so.
- Vulnerabilities affect every single piece of software, so it’s an important threat to take care of them.
- There are different types of vulnerabilities as well as different threat levels.

Bugs vs Vulnerabilities

- Not all bugs are vulnerabilities
    - Unexpected behaviour
- Not all vulnerabilities are bugs
    - Think about backdoors :)

I found a bug!

I found a bug!

- So.. what happens when you found a bug?

You need to understand it, and evaluate if its exploitable or not.

Disclosure Procedure

When a vulnerability is found, the researcher have the responsibility to take an action on this

- Keep it private for personal or friends use
- Sell it, usually you will never know the customer and you own your ownership
- Participate in bug bounties
- Use platforms like hackerone
- Contact the company/project developers
- Publish it anonymously
- Tweet it, with full ownership and recognition

Disclosure Procedure

- It is very delicate to take a decision here and it depends a lot on the kind of target and how the company and users are affected.
- Economical benefit is usually an option
- But reporting it to the company can help them fix it and make their users safe without a pain.
- Any of those options (outside a bounty program) is dangerous because the company can take legal actions against you.
- Minds change and bugs are taken more seriously over time, more companies understand this and setup bounty programs

Disclosure Procedure

- The responsible disclosure consists on reporting it to the company and getting assigned a CVE
- This will let you define some time frames to proceed with the disclosure and fix of the vulnerability.
- This time can be from days to months. And some contracts may restrict you from making any details public.
- It depends on how easy is the update and how critical the bug is.

What’s a CVE?

- Stands for common Vulnerabilities and Exposures
- Public record of vulnerabilities indexed by time

Visit and query some CVEs

- https://www.cvedetails.com/
- https://cve.mitre.org/find/
- https://nvd.nist.gov/vuln/search
- ...

Types of bugs

Which types of vulns exist?

There are many types of vulnerabilities. To illustrate that we will go thru the CVE website and find one of each type.

- Denial of Service
- Remote Code Execution
- Memory corruption
- Sandbox escape
- SQL Injection
- Directory Traversal
- XSS
- …

https://www.cvedetails.com/vulnerabilities-by-types.php

Resources

Programs interact with the external world by accessing different resources, in modern systems thru syscalls, or by using special privileged instructions if permitted.

- Filesystem
- Network
- Audio/Video
- …

Everything can be sandboxed, to restrict the capabilities of the program even more than the ones defined by the user.

Denial Of Service

The application or service no longer works, stops responding requests from the user.

- Assert()
- Infinite loops
- Non-exploitable crashes
- Heavy CPU load

Remote Code Execution

- It is possible to execute code after exploiting the vuln
- This can be caused by an overflow, or other methods like shell command injection bugs.
- Doesn’t implies privilege escalation

An RCE can be achieved in many different ways, it does not define the method, just the final action.

Memory Corruption

- When the memory is polluted the application can miss-behave, in case that no code execution is possible.
- This is usually caused by a buffer overflow
    - But can be because of an use after free
    - or a double free
    - or an integer underflow

- This miss-behaviour can result in a visual glitch, trash input fields, etc
- Or… even execute code

Privilege Escalation

- When the sandbox is bypassed
- When being able to run code as a different user
- Access restricted resources
- Got root?

Directory Traversal

- The attacker can use ../ or other special chars to trick the program and read or write things in unexpected places
- This is usually exploited by abusing path filtering checks or using different encodings for the path which leads to trick the verifications of the program.
- Programs like web servers and web applications are usually the most affected by this kind of vulnerabilities.
  - But that’s also happening in many other kind of applications..
  - even r2!

Unsanitized Inputs

- Can produce unexpected results when presented to the user.
- Can escape to the shell
- Execute or inject code as interpreted in a webview
- Change text in screen
- Glitch the render

Race Conditions

- Happen in multiprocess or multithreaded systems which lack proper verification of shared resource access.
- Many libraries provide atomic and thread safe data structures
- Adds a heavy payload that is not desired in many situations.
- Atomic operations are not always possible
- Sometimes hard to reproduce

See Nighterman’s talk at r2con 2017 for kernel details

Techniques

Finding bugs: techniques

There are several techniques to find bugs in programs, libraries, kernels, drivers, cpus..

- Manual testing
- Automated scripts finding common vulns
- Source code analysis
- Dynamic Analysis
- Fuzzing...

Finding bugs: types of crashes

- Once we found a bug we have to analyze the crash (if any) and determine if it’s exploitable or not.

At this point a crash can be:

- Runtime assertion
- Null pointer dereference
- Buffer overflow
- Use after free
- Integer sign
-

Identifying vulnerabilities in the source

There are different tools and products that perform static source code analysis in order to find

- undefined behaviour
- bad development practices
- wrong use of apis
- Oob read
- Memory leaks
- Oob write (buffer overflow)
- Null deref
- Pointer aliasing alignment issues
- ...

Identifying vulnerabilities in the source

Opensource tools

- Clint
- Clang-analyzer aka scan-build
- Asan
- Gcc warnings

ClosedSource

- Coverity
- ...

Static code source analysis tools

Those tools use to require language specific features to work properly. Each language have its own tools and testsuites to find bugs too.

Let’s try clang-analyzer on any random project

Clang-Analyzer

Compiler plugin for clang that find bugs in supported languages, mainly C and C++.

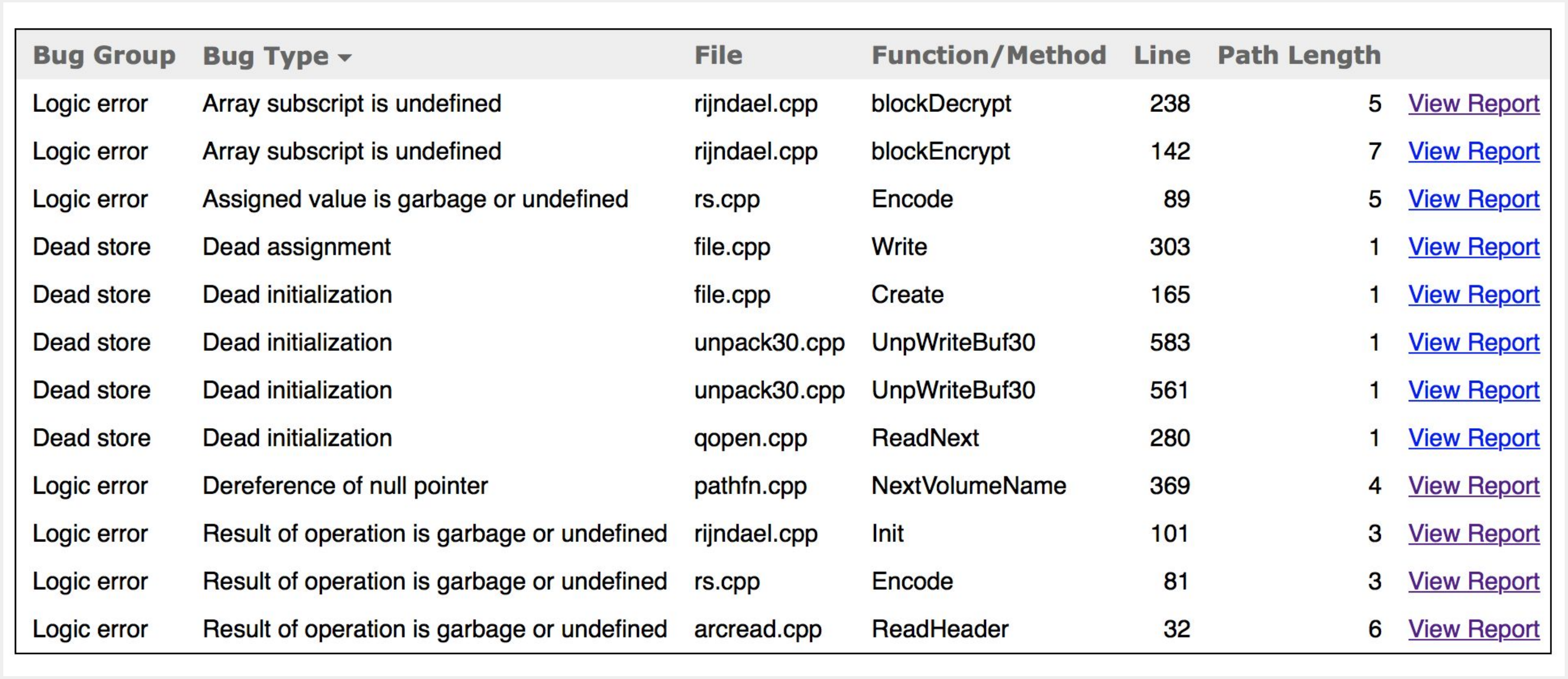
Usually available in your distro of choice

But there are official binary builds for mac and linux.

- sys/clang-analyzer.sh

Clang-Analyzer

- For unrar, we get a report like this. It’s all true?



Bug Group	Bug Type ▾	File	Function/Method	Line	Path Length	
Logic error	Array subscript is undefined	rijndael.cpp	blockDecrypt	238	5	View Report
Logic error	Array subscript is undefined	rijndael.cpp	blockEncrypt	142	7	View Report
Logic error	Assigned value is garbage or undefined	rs.cpp	Encode	89	5	View Report
Dead store	Dead assignment	file.cpp	Write	303	1	View Report
Dead store	Dead initialization	file.cpp	Create	165	1	View Report
Dead store	Dead initialization	unpack30.cpp	UnpWriteBuf30	583	1	View Report
Dead store	Dead initialization	unpack30.cpp	UnpWriteBuf30	561	1	View Report
Dead store	Dead initialization	qopen.cpp	ReadNext	280	1	View Report
Logic error	Dereference of null pointer	pathfn.cpp	NextVolumeName	369	4	View Report
Logic error	Result of operation is garbage or undefined	rijndael.cpp	Init	101	3	View Report
Logic error	Result of operation is garbage or undefined	rs.cpp	Encode	81	3	View Report
Logic error	Result of operation is garbage or undefined	arcread.cpp	ReadHeader	32	6	View Report

Building with ASAN

- ASAN is a compiler plugin and library that adds minimal overhead to the executables to instrument memory corruptions and other vulnerabilities.
- It is not supposed to be used for production systems, but instead, help bughunters and developers.
- Supports gdb in the sense that does early stop when the overflow happens for proper inspection

```
$ sys/asan.sh
```

Testing ASAN

Check the sources/ for:

- yara, r2, binutils, unrar, zzuf, ...

Dynamic Testing

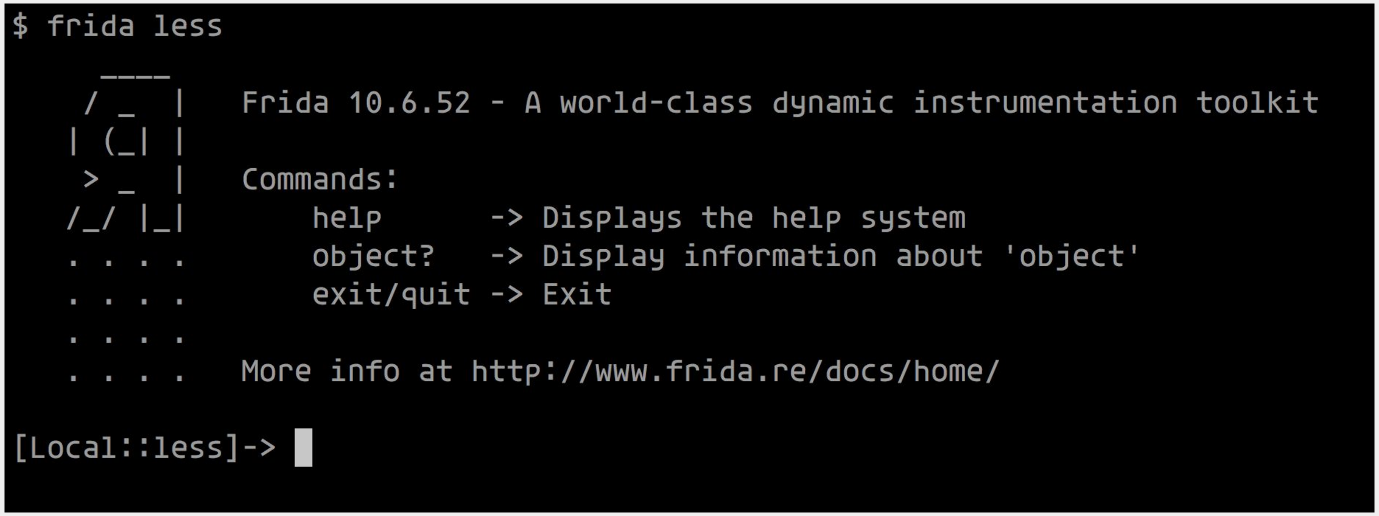
Dynamic Analysis

There are different tools that perform runtime analysis of a process in order to track memory use, guard specific functions, hook methods, etc

- Pin
- DynamiRIO
- Frida
- Valgrind
- Dtrace
- Gperf

Dynamic Analysis

- Most of those tools require manual work to specify what do you exactly want to trace.
  - Interactive with a repl or ui
  - Predefined with a config file



```
$ frida less
     ____
    / _  |   Frida 10.6.52 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[Local::less]->
```

Running with Valgrind

- Incompatible with asan
- Does runtime instrumentation, doesnt requires access to the sources.
- Runs natively but with heavier payload
- Supports x86 and arm, mainly for Linux
- Supports plugins for profiling, memleaks, etc

Binary Static Analysis

Performing static analysis on binaries is usually a complex task, which use to require manual intervention to refine the results.

- We have to define the targets that we want to find
- Create scripts, signatures or templates to find them
- Not all bugs can be found statically
- Many times we will need to do emulation
- The compilation process removes information from the source, so we lack all the info available there but we have a different perspective of the problem.

Disassembly of the Bugs

Let’s see how a bug or a vulnerability looks in source and bin

Identifying dangerous libc functions

- What is a dangerous libc function?

Functions from posix that has been marked about problematic for a bunch of reasons.

- Developers don’t what they are doing
  - Memcmp vs strncmp
- Inconsistency of size arguments with null terminators
  - Strncpy vs snprintf
- Filling user provided buffer without given size
  - FGets, …
- Some format strings can result in crash or vuln

Identifying vulnerable statically linked libs

We can write yara rules to check if some strings are inside the binary.

There are also binary signatures with r2 and ida using flirt or in r2 with its own format.

Finding Them

Usually those functions are found in the libc. We may find other unrecommended, or dangerous functions in different libraries too.

If we are lucky and the bin is not statically linked we should just check the imports.

Analyzing how are they used

When we found the import, we have to find the xrefs and see how those functions are called.

- We should identify if the arguments are correct
- Check if the return value is used or ignored
- Verify pre and post conditions for each argument
    - Boundaries
- Extra checks depending on the type information

Signatures to find statically linked libs

There are many ways to identify which libraries has been statically linked.

- Checking strings
- Checking for function signatures

Function signatures are usually byte patterns with binary mask

But can be improved by

Example recovering symbols from a stripped Go program

Which Libraries are statically linked?

Take one binary and find out which libraries are statically linked inside.

- Use google for the strings
- Try to identify the version number
- Can we identify the username of the developer?
- Can we know the date of the build?
- ...

Finding Vulnerabilities in binaries

-

Which functions are easier to have bugs?

- Reentrant (stack exhaustion)
- Basic block graph complexity
- Calling dangerous or unsafe functions
- Processing data from external sources
- Printing data from external source without sanitizing
- ...

Solutions

- Run the code in a vm
- Run the code sandboxed
- Use source and binary analysis tools
- Use safe or dynamic languages which aim to abstract all the unsafe logic that can cause memory corruption bugs.
- Nothing saves from logic bugs
- Race conditions are hard to debug, but there are tools

CALEA

- What it is?
- Where it is used?
- What’s wrong with it?

Blindly finding vulns

- Sometimes we have no way to run the application of interest in local, and sometimes we can’t attach to it.
- Crash logs are sometimes not available
- So we find unexpected results by testing different inputs.
- No answer or the service being shutted down are common behaviours when the app has crashed.

Crash Logs

- Many Operating Systems generate a crash log when
- There are application frameworks that implement the same functionality.
    - Handles crashing exceptions or signals
    - Generates a report with a report for all the threads
        - Containing backtrace
        - Memory maps
        - Registers
        - Disasm around $pc

Core Files

- Contain the same information as the crash log
- But adds all the data in the stack and the heap.
- This is, the user can have a post-mortem snapshot of what happened at the moment of the crash.

Time Machine!

- Wouldn’t be great if we could travel back in time and find out what happened right before the crash?
- We have some tools that can record an execution
  - QIRA
  - RR
- Some debuggers support this functionality by saving and restoring the state and the changes of every memory snapshot.
  - Gdb, r2, ida (replay debugger), ..

Reverse Debugging

- Example using reverse debugging with r2 or gdb.

```
(gdb) record; stepi; reverse-stepi

(r2) dts+;ds;dsb
```

Understanding The Crash

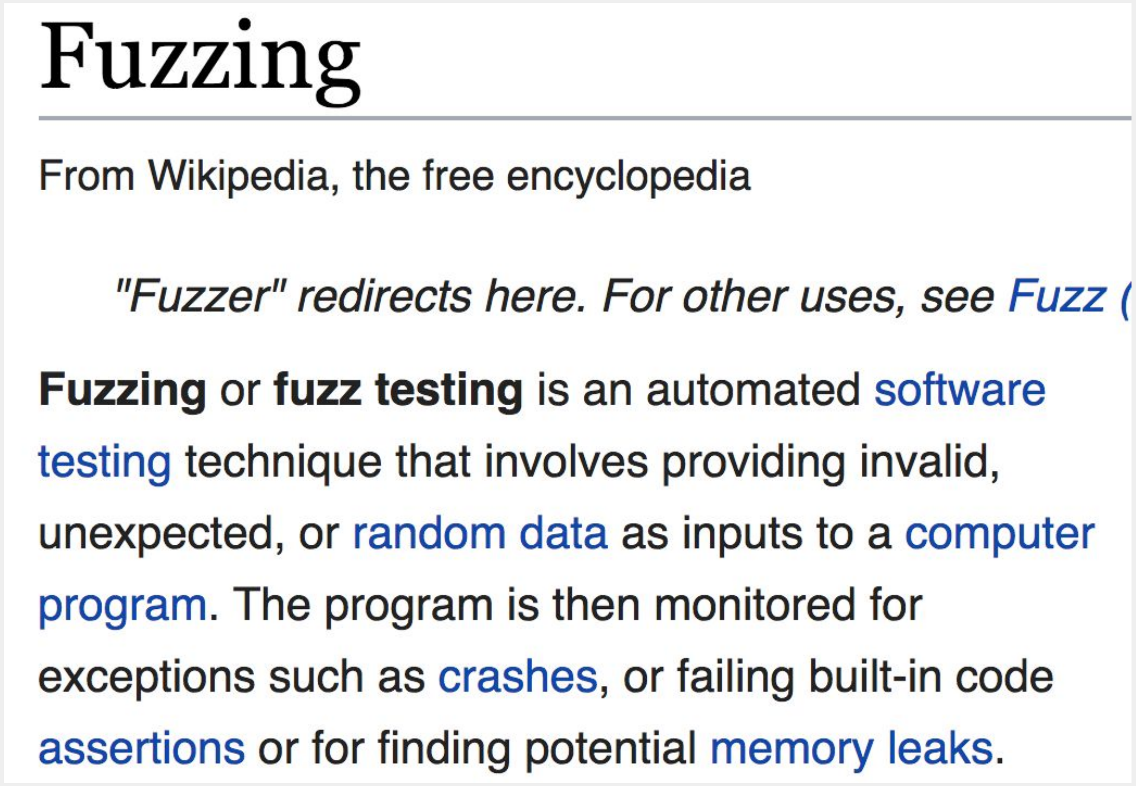
- Check the register values
- Check the backtrace
- Check the disassembly involved

- It is exploitable?

Fuzzing

What’s fuzzing?

https://en.wikipedia.org/wiki/Fuzzing



Fuzzing

From Wikipedia, the free encyclopedia

"Fuzzer" redirects here. For other uses, see Fuzz (

Fuzzing or **fuzz testing** is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks.

Generating Samples

- The key of fuzzing is being able to fill all possible execution flows in a program or a specific function of interest.
- In order to achieve this we need to understand the conditionals inside the code and which data from the input causes a branch.
- Generating the most optimal samples to reduce executions requires an extra work on runtime tracing analysis.
- To reduce the amount of useless samples it is important that the generator understands the logic of the expected input which can be a file format, etc

Generating Samples with templates

- The best way to start fuzzing is by having some legit samples and tweak some random bytes and see how the target program behaves.
- Sometimes those changes require checksums to be updated, so without understanding the logic of the format we will miss a lot of rounds.

Zzuf

- This is probably the simplest way to modify a binary and get those small changes applied in a reproducible way.
- Being able to reproduce a sample or a crash is very important for fuzzing. So we can reduce the size of the sample database and reduce reports and allow manual testing later.

```
$ zzuf -s $RANDOM < /bin/ls > ls2
```

Radamsa

- It works similar to zzuf, with the difference that it is able to do changes based on known file formats. So it produce better samples in less time than blind patching.

https://github.com/aoh/radamsa

```
$ echo '((lambda (x) (+ x 1)) #x124214214)' | radamsa -n 10000
```

AFL

- The American Fuzzy Lop
- Probably the smartest fuzzer out there
- Developed by taviso (head security guy at google)
- They use it in the Google Cluster Fuzz
- Mixes dynamic instrumentation, code analysis, runtime information in order to produce better input to cover all possible code paths.
- Usually takes more time than radamsa or zzuf to start getting results but it pays off in the long term

Classification

Classification of crashes

- When we are fuzzing, we use to require:
  - large amount of ram, to spawn more processes in parallel
  - A lot of cpu, to generate more samples per second
  - More disk to store all those samples

So we end up generating a large set of samples, which some of them will generate crashes or infinite loops.

- Stop execution after some time
- Store all the crash logs in a parseable format
- Remove false positives
- Purge duplicates

How to know if it’s exploitable?

- Can we get control of the program counter?
- Can we pivot the stack?
- Can we write anywhere? Heap

Nightmare Frontend

- Nightmare is a fuzzer frontend
  - Written in Python
  - Web interface to run the service
  - Supports multiple fuzzers
  - Generate reports and statistics of crash logs
  - Supports multiple parallel executions

Finding 1-days

Security Updates

- When a vulnerability is published, vendors deliver patches and attach some notes showing details on how critical this update is and the security implications.
- A well known technique for analyzing vulnerabilities and understanding the patches is by checking for the changes between the old and new version of a specific library or program after having a security update.

Diffing Versions

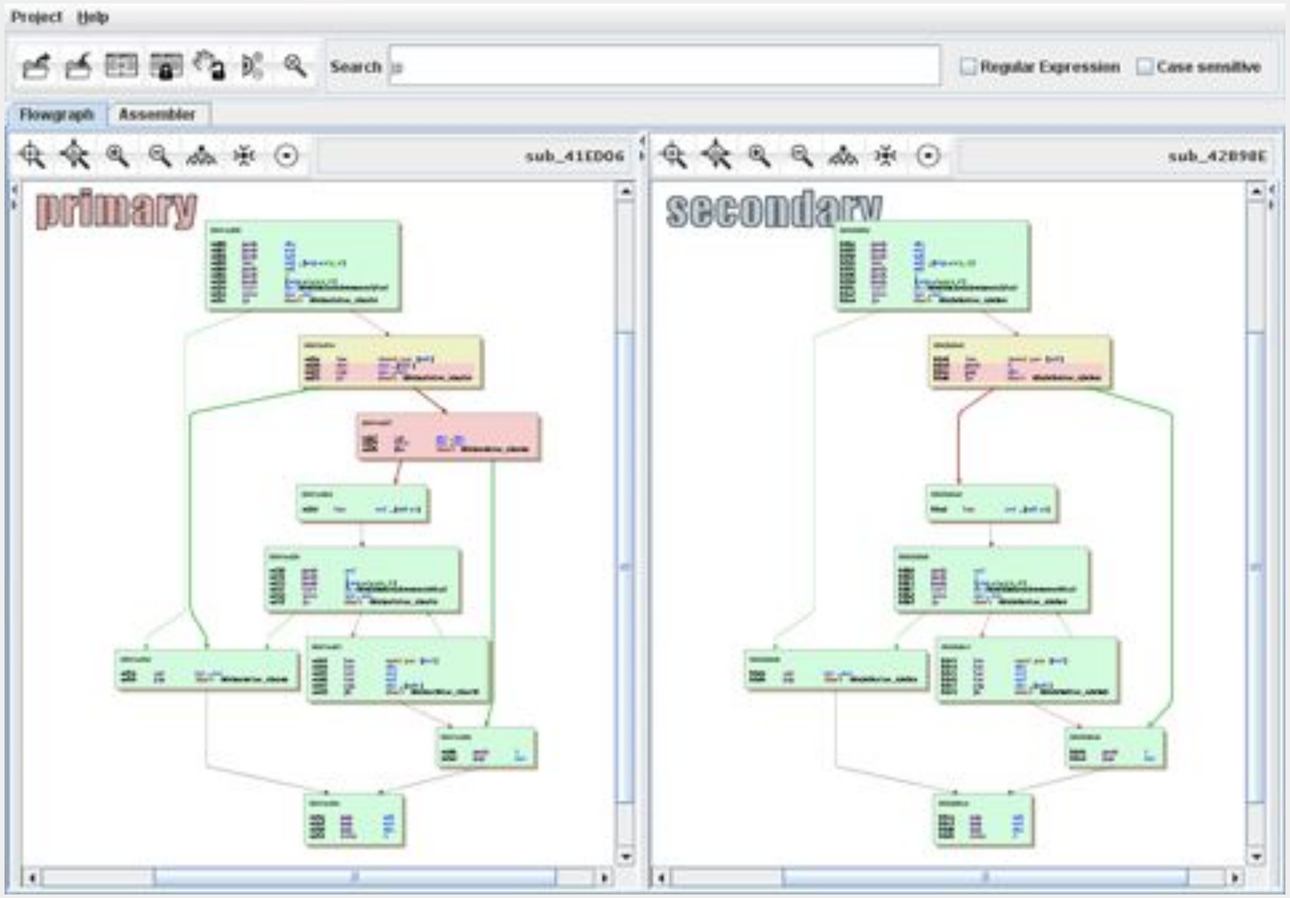
- Many people write blog posts analyzing the changes between the old and new affected binaries, we can google for them.
- To ease the task of learning how to bindiff a vulnerability, we can start by checking open source projects and taking binary builds as input for our tests.
- We can later check the related git commit.
    - Grep for ‘fix’, ‘segfault’, ‘overflow’
    - Important projects use to hide those keywords in commit msgs

Diffing Tools

- There are many tools to find the differences between two binaries, by understanding some logic inside the file.
    - Parsing Headers
    - Analyzing symbols
    - Find functions
    - Generate call-graphs
    - Compare disassemblies
    - Remove variable stuff (offsets, addresses, …)

Bindiff

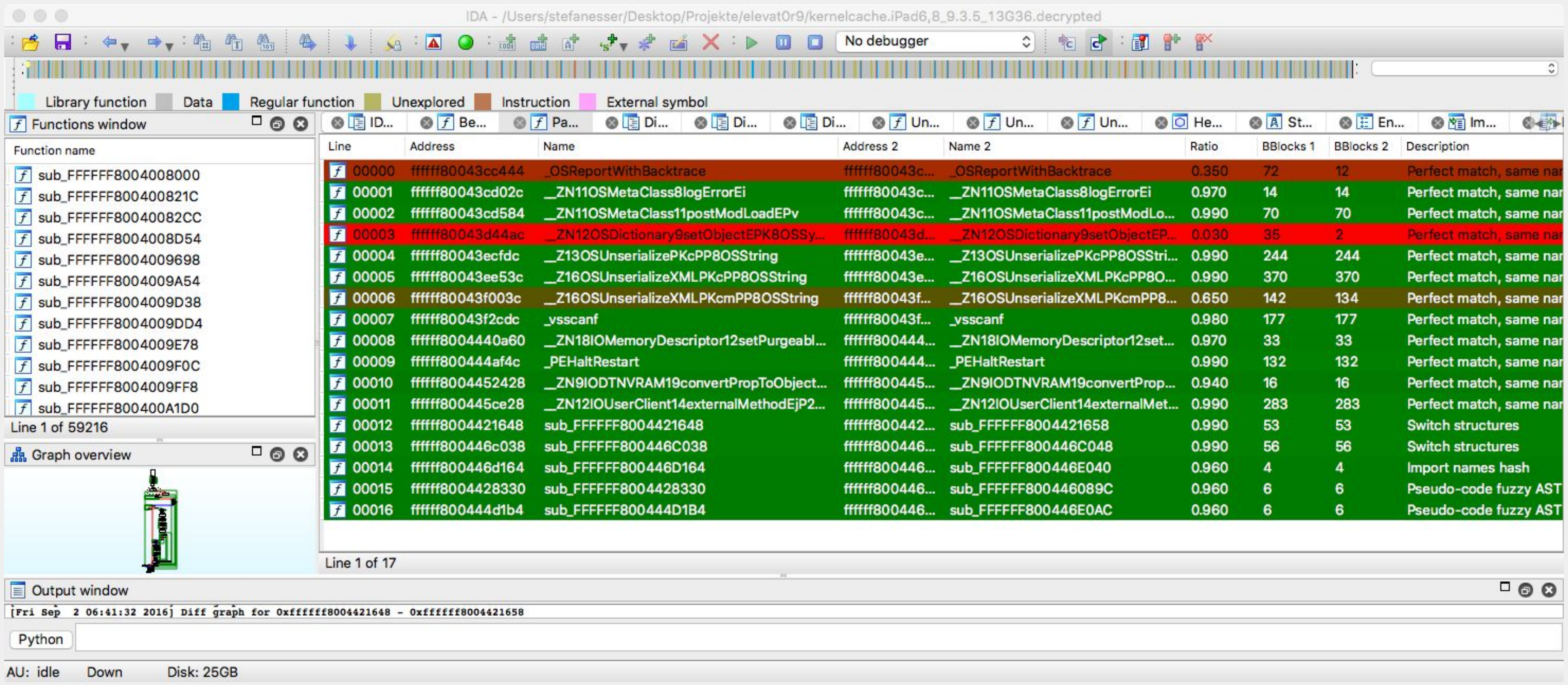
- Unmaintained, but a reference in the scene



Diaphora

- Developed by Joxean Koret

https://github.com/joxeankoret/diaphora


IDA - /Users/stefanesser/Desktop/Projekte/elevat0r9/kernelcache.iPad6,8_9.3.5_13G36.decrypted

No debugger

Library function | Data | Regular function | Unexplored | Instruction | External symbol

Functions window

Function name
sub_FFFFFF8004008000
sub_FFFFFF800400821C
sub_FFFFFF80040082CC
sub_FFFFFF8004008D54
sub_FFFFFF8004009698
sub_FFFFFF8004009A54
sub_FFFFFF8004009D38
sub_FFFFFF8004009DD4
sub_FFFFFF8004009E78
sub_FFFFFF8004009F0C
sub_FFFFFF8004009FF8
sub_FFFFFF800400A1D0

Line 1 of 59216

Graph overview

ID... | Be... | Pa... | Di... | Di... | Di... | Un... | Un... | Un... | He... | St... | En... | Im...

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00000	ffffff80043cc444	_OSReportWithBacktrace	ffffff80043c...	_OSReportWithBacktrace	0.350	72	12	Perfect match, same nar
00001	ffffff80043cd02c	__ZN11OSMetaClass8logErrorEi	ffffff80043c...	__ZN11OSMetaClass8logErrorEi	0.970	14	14	Perfect match, same nar
00002	ffffff80043cd584	__ZN11OSMetaClass11postModLoadEPv	ffffff80043c...	__ZN11OSMetaClass11postModLo...	0.990	70	70	Perfect match, same nar
00003	ffffff80043d44ac	__ZN12OSDictionary9setObjectEPK8OSSy...	ffffff80043d...	__ZN12OSDictionary9setObjectEP...	0.030	35	2	Perfect match, same nar
00004	ffffff80043ecfdc	__Z13OSUnserializePKcPP8OSString	ffffff80043e...	__Z13OSUnserializePKcPP8OSStri...	0.990	244	244	Perfect match, same nar
00005	ffffff80043ee53c	__Z16OSUnserializeXMLPKcPP8OSString	ffffff80043e...	__Z16OSUnserializeXMLPKcPP8O...	0.990	370	370	Perfect match, same nar
00006	ffffff80043f003c	__Z16OSUnserializeXMLPKcmPP8OSString	ffffff80043f...	__Z16OSUnserializeXMLPKcmPP8...	0.650	142	134	Perfect match, same nar
00007	ffffff80043f2cdc	_vsscanf	ffffff80043f...	_vsscanf	0.980	177	177	Perfect match, same nar
00008	ffffff8004440a60	__ZN18IOMemoryDescriptor12setPurgeabl...	ffffff800444...	__ZN18IOMemoryDescriptor12set...	0.970	33	33	Perfect match, same nar
00009	ffffff800444af4c	_PEHaltRestart	ffffff800444...	_PEHaltRestart	0.990	132	132	Perfect match, same nar
00010	ffffff8004452428	__ZN9IODTNVRAM19convertPropToObject...	ffffff800445...	__ZN9IODTNVRAM19convertProp...	0.940	16	16	Perfect match, same nar
00011	ffffff800445ce28	__ZN12IOUserClient14externalMethodEjP2...	ffffff800445...	__ZN12IOUserClient14externalMet...	0.990	283	283	Perfect match, same nar
00012	ffffff8004421648	sub_FFFFFF8004421648	ffffff800442...	sub_FFFFFF8004421658	0.990	53	53	Switch structures
00013	ffffff800446c038	sub_FFFFFF800446C038	ffffff800446...	sub_FFFFFF800446C048	0.990	56	56	Switch structures
00014	ffffff800446d164	sub_FFFFFF800446D164	ffffff800446...	sub_FFFFFF800446E040	0.960	4	4	Import names hash
00015	ffffff8004428330	sub_FFFFFF8004428330	ffffff800446...	sub_FFFFFF800446089C	0.960	6	6	Pseudo-code fuzzy AST
00016	ffffff800444d1b4	sub_FFFFFF800444D1B4	ffffff800446...	sub_FFFFFF800446E0AC	0.960	6	6	Pseudo-code fuzzy AST

Line 1 of 17

Output window

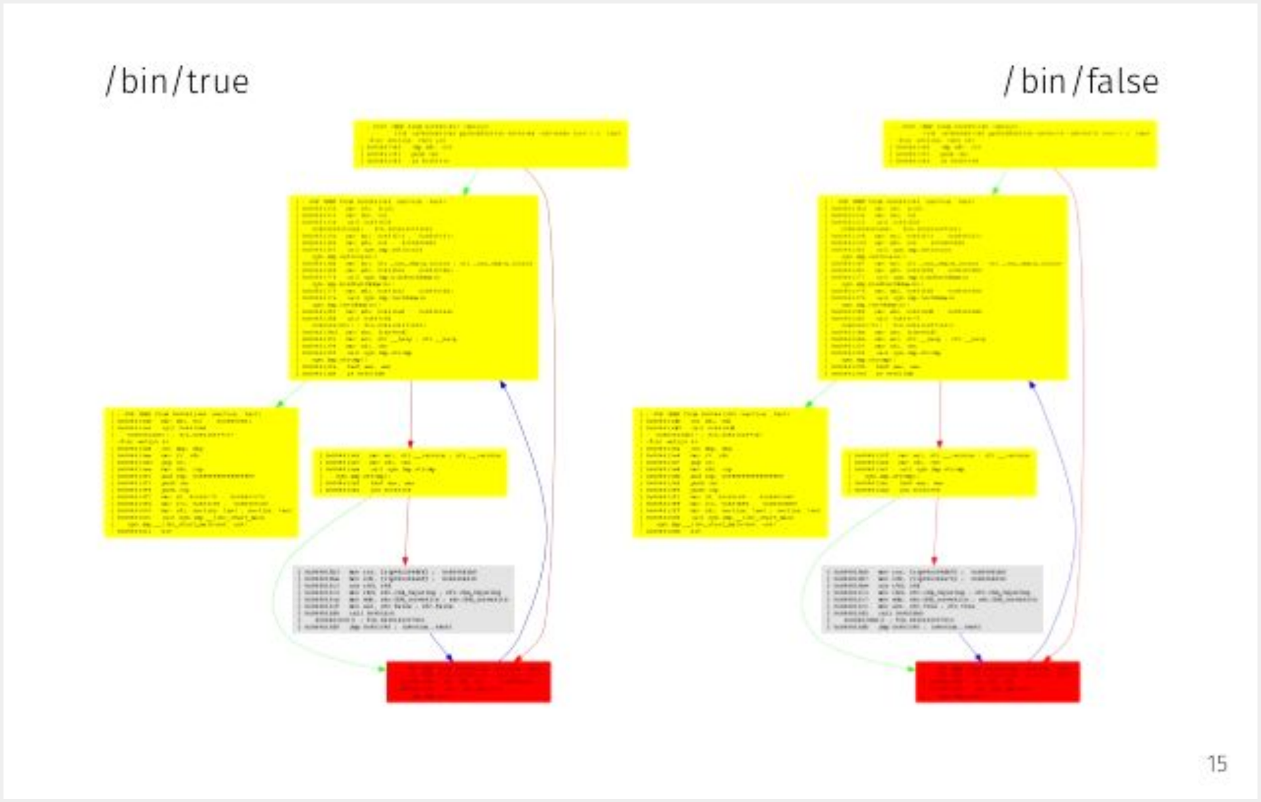
```
[Fri Sep  2 06:41:32 2016] Diff graph for 0xffffff8004421648 - 0xffffff8004421658
```

Python

AU: idle   Down   Disk: 25GB

Radiff2 (From radare2)

- radiff2 -g



Healing bins

Fixing vulnerabilities without the source

Many times we find a crash but we don't have access to the source.

We can fix this bug in different ways:

- Injecting a library
- Tracing the execution
- Binary patching

The choice may depend on our needs

Injecting a library

Learn to use LD_PRELOAD

replace a symbol from libc to avoid overflow

- strcpy
- gets
- sprintf
- strcat
- ...

Tracing the execution

- Let’s use Frida to hook an import and change its behaviour to avoid the app to crash.
- Same as we did with LD_PRELOAD

The tool ‘frida-trace’ allows us to easily trace all symbols and imports of a program:

```
$ frida-trace -i ‘*’ /bin/ls
```

```
$ vim __handlers__/*
```

Binary Patching

We will use r2 in write mode to fix the vuln. There are several ways to solve this:

- Increase the stackframe
- Rewrite the function code
- Call intermediate proxy function
- Find room to write this code
- Maybe we can just nop it? Not for this case
- ...

0patch

This project provides on-the-fly runtime security binary patches to be applied to Linux and Windows kernels.

- Solves the reboot problem (SLA)

Fixing a vulnerability in the binary

Demo with r2

CrashLogs

Crashlogs

- Parsing is not easy because format may change
- Designed to be read by humans:
    - Register state and backtrace of every thread
    - Exception information and offending thread id
- Let’s see some examples of crash logs

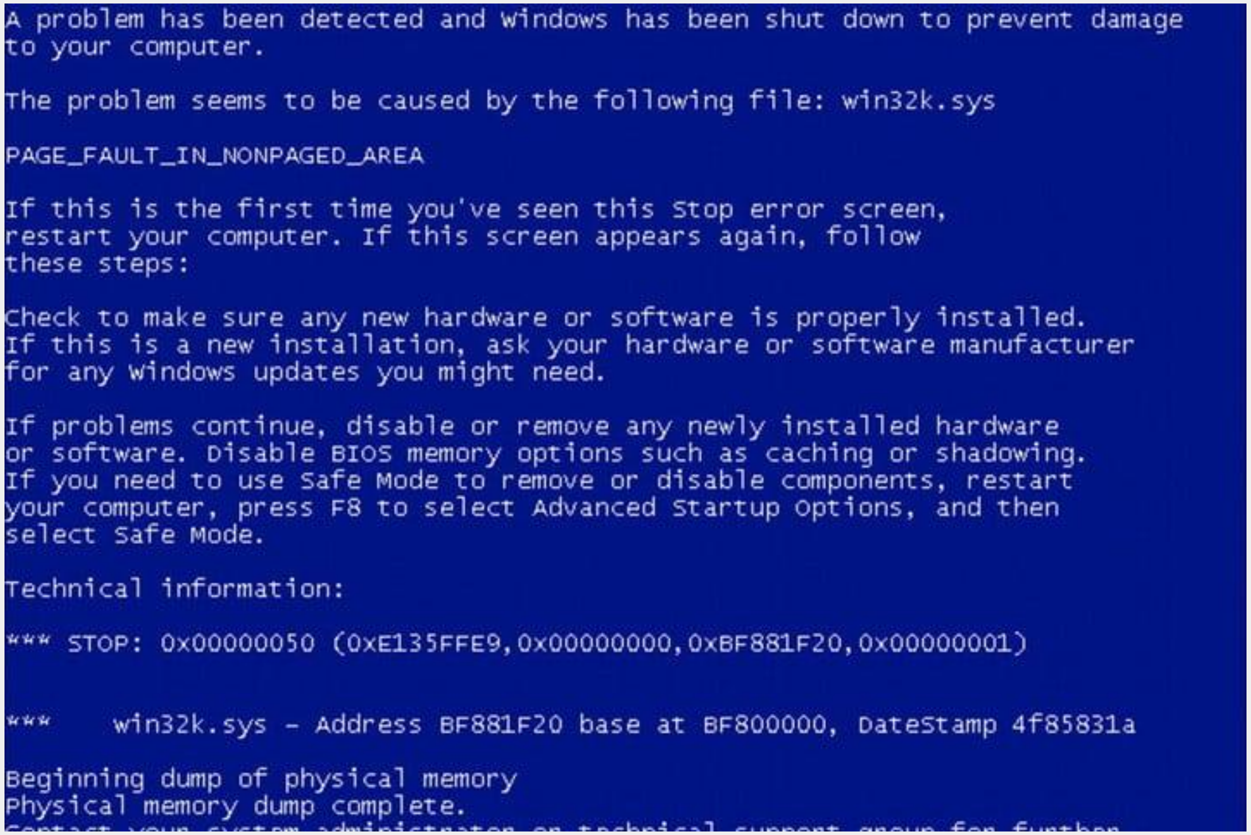
Just In Time Debuggers

Mac, Linux and Windows supports this functionality

- At the moment of crash, the application is halted waiting for a debugger to be attached
- The debugger of choice is spawned and generates a report
- The report may contain different information
    - Serial number. Useless for users, meaningful for the developers
    - Obfuscated crash logs
    - Regstate + for all threads
    - Memory map
    - Backtrace of all threads
    - ...

Crash Logs: Microsoft

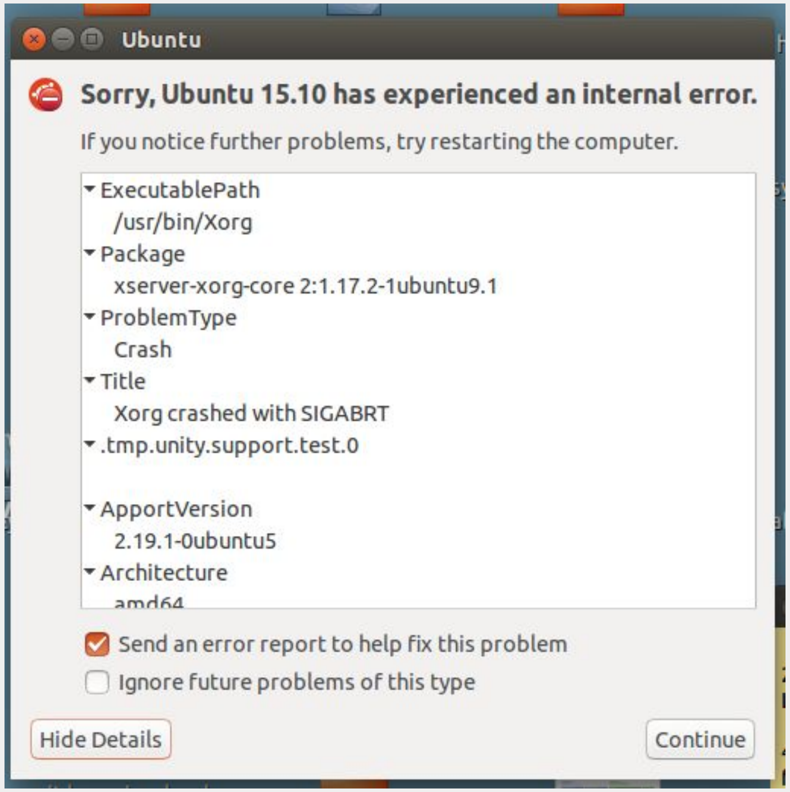
- The famous Microsoft blue screen of death



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: win32k.sys

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xE135FFE9,0x00000000,0xBF881F20,0x00000001)


***    win32k.sys - Address BF881F20 base at BF800000, DateStamp 4f85831a

Beginning dump of physical memory
Physical memory dump complete.
```

Crashlogs: Ubuntu

- Ubuntu provides a pretty readable crashlog
- Submit button sends the issue to Ubuntu.
- Other distros have similar approach
- As well as standalone apps, which some of them have crash handlers to report stuff to the authors.


Ubuntu

Sorry, Ubuntu 15.10 has experienced an internal error.

If you notice further problems, try restarting the computer.

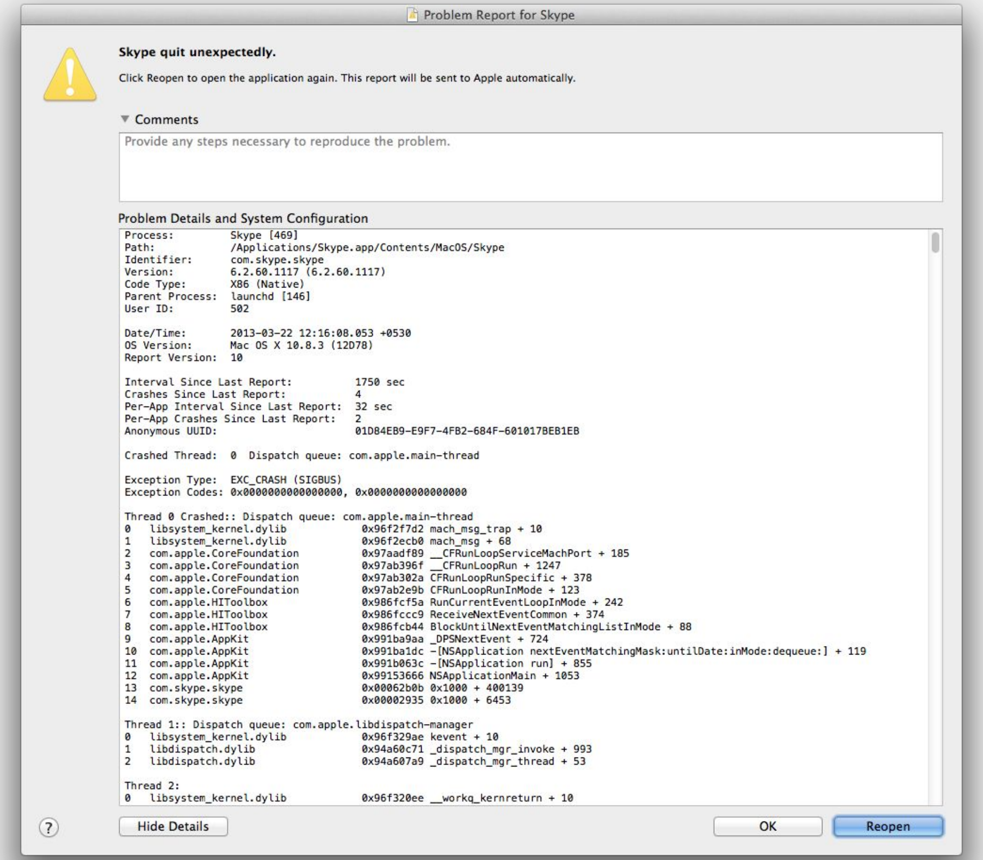
```
▾ ExecutablePath
    /usr/bin/Xorg
▾ Package
    xserver-xorg-core 2:1.17.2-1ubuntu9.1
▾ ProblemType
    Crash
▾ Title
    Xorg crashed with SIGABRT
▾ .tmp.unity.support.test.0

▾ ApportVersion
    2.19.1-0ubuntu5
▾ Architecture
    amd64
```

[✓] Send an error report to help fix this problem

[ ] Ignore future problems of this type

Crash Logs: Apple

-



Skype quit unexpectedly.

Click Reopen to open the application again. This report will be sent to Apple automatically.

Problem Details and System Configuration

```
Process:         Skype [469]
Path:            /Applications/Skype.app/Contents/MacOS/Skype
Identifier:      com.skype.skype
Version:         6.2.60.1117 (6.2.60.1117)
Code Type:       X86 (Native)
Parent Process:  launchd [146]
User ID:         502

Date/Time:       2013-03-22 12:16:08.053 +0530
OS Version:      Mac OS X 10.8.3 (12D78)
Report Version:  10

Interval Since Last Report:          1750 sec
Crashes Since Last Report:           4
Per-App Interval Since Last Report:  32 sec
Per-App Crashes Since Last Report:   2
Anonymous UUID:                      01D84EB9-E9F7-4FB2-684F-601017BEB1EB

Crashed Thread:  0  Dispatch queue: com.apple.main-thread

Exception Type:  EXC_CRASH (SIGBUS)
Exception Codes: 0x0000000000000000, 0x0000000000000000

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   libsystem_kernel.dylib        0x96f2f7d2 mach_msg_trap + 10
1   libsystem_kernel.dylib        0x96f2ecb0 mach_msg + 68
2   com.apple.CoreFoundation      0x97aadf89 __CFRunLoopServiceMachPort + 185
3   com.apple.CoreFoundation      0x97ab396f __CFRunLoopRun + 1247
4   com.apple.CoreFoundation      0x97ab302a CFRunLoopRunSpecific + 378
5   com.apple.CoreFoundation      0x97ab2e9b CFRunLoopRunInMode + 123
6   com.apple.HIToolbox           0x986fcf5a RunCurrentEventLoopInMode + 242
7   com.apple.HIToolbox           0x986fccc9 ReceiveNextEventCommon + 374
8   com.apple.HIToolbox           0x986fcb44 BlockUntilNextEventMatchingListInMode + 88
9   com.apple.AppKit              0x991ba9aa _DPSNextEvent + 724
10  com.apple.AppKit              0x991ba1dc -[NSApplication nextEventMatchingMask:untilDate:inMode:dequeue:] + 119
11  com.apple.AppKit              0x991b063c -[NSApplication run] + 855
12  com.apple.AppKit              0x99153666 NSApplicationMain + 1053
13  com.skype.skype               0x00062b0b 0x1000 + 400139
14  com.skype.skype               0x00002935 0x1000 + 6453

Thread 1:: Dispatch queue: com.apple.libdispatch-manager
0   libsystem_kernel.dylib        0x96f329ae kevent + 10
1   libdispatch.dylib             0x94a60c71 _dispatch_mgr_invoke + 993
2   libdispatch.dylib             0x94a607a9 _dispatch_mgr_thread + 53

Thread 2:
0   libsystem_kernel.dylib        0x96f320ee __workq_kernreturn + 10
```

Exercise: Parsing a Crashlog

- Write a script in any language to parse a crashlog and load this information inside r2
- Optionally extend r2crashlog to let your efforts to be reused by others

```
$ r2pm -i r2crash
```

Questions?