Some technical & scientific challenges I'd like to have working solutions for

Fred "pappy" Raynal
fraynal@quarkslab.com
@fredraynal

QUARKSLAB
INNOVATIVE SECURITY

Who am I?

- Born: 17th March 1972
- 1st computer around 10: a ZX81 with 1KB of RAM
- Bac C in 1990 (now called S)
- Engineering degree + Military service + DEA (Master) in 1996-1997
- PhD. in 2001 at INRIA (wrote articles for LinuxMag at the same time)
- Started MISC, a French security related magazine, in Jan. 2002 - stopped 13 years later!
- Started SSTIC, a French security conference, in Jun. 2002 - stopped 4 years later!
- Co-created and co-run EADS CCR (IW) until 2006
- Created and run Sogeti ESEC R&D until 2011
- CEO and founder of Quarkslab since then

Preamble

- Some technical challenges I'd like to have **working** solutions for
- Working does not mean:
  - 100% efficient or reliable (f*** corner cases)
  - I don't care for technical / scientific elegance
- Working means:
  - Scalable: yes, size matters
  - Answering at least some parts of the problem

Roadmap

- Now classic (but not working):
  - IR, malware analyzis & classification
    - Vulnerability: research, exploitation, …
- What I also need:
  - Obfuscation
  - Patching

Friends will be friends: Incident response

What is Incident Response?

- Incident response is an organized approach to addressing and managing the aftermath of a security breach or attack (also known as an incident).

- The goal is to handle the situation in a way that limits damage and reduces recovery time and costs.

Why do we care?

- Breaches are caused by exploits and “malware”
  - rootkit, spear phishing, virus, APT, …
- So many different kinds of “malware”, we need to automatize
- Attack noise: random attacks not often advanced …
  - but could be used to hide a more serious threat
- “Targeted” attacks: attacks crafted for a specific target
  - Does not mean attach vectors are exclusively used

Incident response in 2 questions

- Is this a threat?

- If yes, how to fix it?

Dealing with incident response

- A 4-stage process:
  - Init: setup tools and knowledge base (learning)
  - Update: keep learning with new binaries / processes but real-time now
  - Alert: the questions one asks when a threat is detected
  - Remediate: fix and prevent

The false problem: unpacking

- Packing: yes, malware can be packed, but unpacking in easy
    - Why unpack? To be able to perform static analyzis
    - Simple tricks are good enough to unpack most of the malware
        - unpacking library
        - kernel instrumentation, non public hypervisor

The problem nobody cares about: *morphism

- Less than 2% of malware use polymorphism
- None known to use metamorphism
- It does not mean it won't one day, but we already have enough sh*t to deal with "basic" and "advanced" threats
  - The day it will be useful, analyzis will be made manually at first

Detection?

- Detection is not a problem anymore => rely on many unreliable tools
    - Specific tools: AV, sandbox, hash database
    - Side channels: big data (DNS, proxy, url, …), behavioural analyzis
- Accept you are going to be compromise by “something” “some day”
    - Be ready to fix quick and clean

Stage #1: init

- Classify files (not only binaries)
  - 3 classes: clean, malware, unknown
- Howto:
  - Human analyzis: expensive and super-slow
  - Similarities: does it really scale?
- The KISS trick:
  - Use a hash database of the users and servers masters

Stage #2: update

- A new file arrives, how to classify it?
  - Clean, malware, unknown
- Usually 2 strategies:
  - Static: when the file is created
  - Dynamic: when the file is executed
- Scaling again: mail, software updates, download, …

Stage #3: alert

- Quickly needs answer to:
  - Q1: Where is the threat?
    - Can use IOC for that, if one has a way to run that on all the attack surface

Stage #4: remediate

- Quickly needs answer to:
  - Q2: How to clean it?
    - Can I “clean” the threat or should I reinstall everything?
    - From a static / dynamic analyzis, can I provide a way to clean the malware?
- Learn from attacks: create new signature / IOC / *
  - Anything helping to detect this attack again

Conclusion

- Dealing with many files: don't work only on malware samples
  - Legitimate, created on the fly, malware, documents, …
- Dealing with huge binaries: use Acrobat Reader as the new Lena
  - Many functions, obviously even more basic blocks
- Any solution not scaling with these 2 constraints can not apply
  - Yes, I prefer to have less matching but to have it working on my file stream


Another one bites the dust:
vuln & exploits

Vulnerability: from bug to exploit

- Every software has bug(s)
- Bugs might have consequences:
  - DoS: a simple crash, memory exhaustion, …
  - Code exec: change the legitimate control flow of the target program
- Won’t explain why we care for bugs & exploits

Bugs & Exploits

WARNING

Bug research != Exploit development

Strategies to find bugs: hand-made

- Hand-made: sources | binary analyzis
  - Allows to find the best bugs …
    - Including the logic ones
  - … but so time consuming
    - how to: top / down, bottom / up, bug types

Strategies to find bugs: half-automatic

- Half-automatic: help to locate points of interest
    - Diffing: if there was a bug there before, another one might still be around
        - Good opportunities vs. difficult context analyzis
    - Source analyzis: look for vuln in a code
        - Soundness vs relevance issues + hard to setup
    - Fuzzing: send malformed inputs to a program
        - KISS vs difficult context analyzis

Strategies to find bugs: automatic

- Automatic: mostly abstract interpretation to prove code
  - AI proves a class of bug is absent in a code
  - But if it fails, is it because:
    - An over-approximation or a real bug?
    - Because the code does not comply to the prover
- Not relevant for bug finding because:
  - Barely scales on “real” code (embedded / critical systems only)
  - Almost impossible to use without 3 PhDs

Exploitation

- No more automatic methods to exploit
- Don’t necessarily need a detailed analyzis of the root cause
- Need to understand the memory map:
  - Local variables, registers, internal structures
  - Controlled memory or not
  - Context execution: ASLR, DEP, stack cookie, …
- Today’s exploits are craftsman work

Source audit is the old black

Source code tools

- Cons
  - Many are VERY expensive
  - Requires to have 3 PhDs to be able to use one
  - Relies on non-trendy language (O-Caml)
- Pros
  - Would love to find some but never used one of these tools …

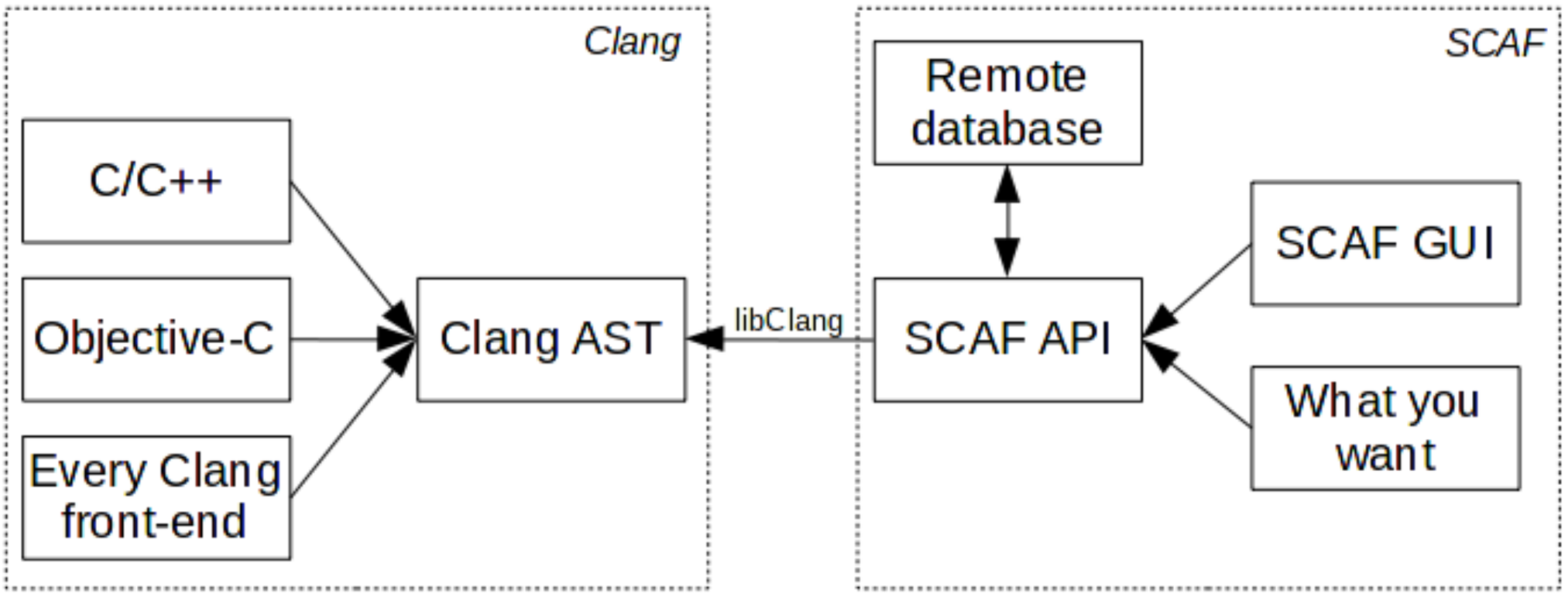
Handmade source audit

- What to look for: kind of signatures
    - Functions (ex.: memory handling), pointer arithmetic, bug class (integer overflow), …
- How to look for
    - Top / down : start from the main() and move down to the leaves
    - Bottom / up : start from a leaf and get backward to the main()
    - Focused: a specific class (ex.: strings), feature (ex.: a parser)

Our tool^w PoC: SCAF

- Wanted killer features
  - Contextualizition: declaration, xref, CFG, …
  - Framework: want to make plugins and automatize sometimes
  - Collaborative: several users at the same time
  - Scalable: test vector was the linux kernel

SCAF in one picture



Source analyzis

- Important difference
  - Usual tools: look for everything
  - Human analyst: look for one thing
- I don't want tools trying to answer questions I did not even ask
- Good test cases: Linux kernel, openssl, Firefox, image libraries…

Clash of clans

- “Hacker” world: efficiency first, barely strong theory, deal is the current problem the guy is working on

- “Academic”: unusable by a normal human (theory), not user friendly (UI), works on small examples (scalability)

- Conclusion: still a lot of research to do in source analyzis, but should focus on making the current results accessible / usable

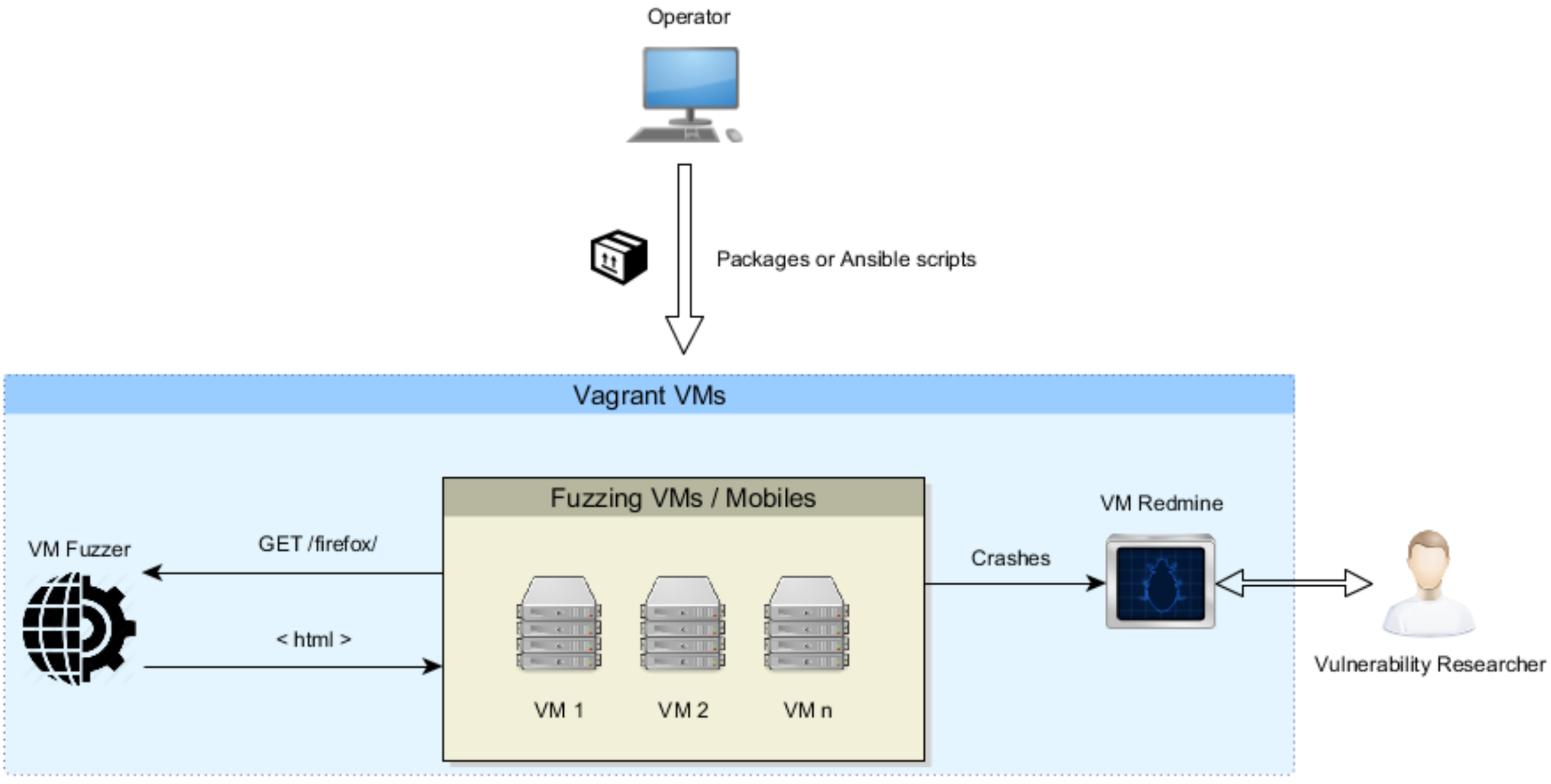
- Bonus: would also love to improve context based on execution traces

Fuzzing is the new black

Fuzzing 101

- Definition (fuzzing): transform sh*it into gold
- Can start very cheap up to a very complex infrastructure
- The best approach today because the best ratio effort / gain

Complex fuzzing infra



(Dumb) Fuzzing 102

- Automatic:
  - Mutation: transform an input into another
  - Agent / supervisor / monitor : keep an eye of the target's behaviour when it is feed with the input
  - Triage: keep record of "interesting" events
- Hand made:
  - Crash reduction: reduce sample to its minimum
  - Crash analyzis: is it exploitable or not?

Problems with fuzzing?

- Code coverage: try to reach every basic block
  - Easy to measure at execution but slows down the execution
  - Greatly depends on non mutated inputs (corpus)
- Data coverage: try to cover the data space
  - Attempt to solve the problem with symbolic / concolic execution to find / solve the constraints
  - Not reliable and super slow
- “Interesting” event: what is a deviant behaviour?

Code coverage at scale

- Corpus distillation
    - Gather a large set of inputs (the corpus)
    - For each, generate an execution trace to store
    - Select the minimal subset in the input covering all the code
    - Start usual dumb fuzzing with this subset as to-be-fuzzed corpus

Corpus distillation numbers

- 20 TB of SWF files
- 1 week of computation on 2000 CPUs to select the minimal corpus => 20 000 files
- 3 weeks of mutation / execution on the 2000 CPUs => 400 unique crashes
- 106 security bugs acknowledged by Adobe
- Source: http://googleonlinesecurity.blogspot.fr/2011/08/fuzzing-at-scale.html

Corpus distillation issues

- Mostly only Google is able to do that because they already have all the files

- Does not really solve the problem, use scale for that … but at least, it works!

  - Also means there is a big research to do there ;)

Afl - American Fuzzer Loop

- Based on edges, not basic blocks anymore
- Focus on transitions rather than instructions
- Target is instrumented either at compilation or execution
- Manage its own corpus through genetic algorithms
- Provide tools to manage crashes, corpus, *
- Best fuzzer today: http://lcamtuf.coredump.cx/afl/

Challenges with fuzzing

- Corpus: have minimal samples with max code | data coverage
- Code coverage: block vs. edge? Static with use of subgraphs?
- Data coverage: symbolic, concolic
- Crash reduction
- Exploitability

The march of the black queen:
Code obfuscation

Obfuscation

- Obfuscation: make programs difficult to analyze
  - Must protect both code and data sometimes
  - Focus very often on the CFG only, not on the data
- Constraint: try not to kill the performances (CPU, memory)

Some techniques

- Opaque predicates, code flattening, inlining / outlining, virtual machines…
- Mixed boolean-arithmetic expressions, string encryption, parameters or prototypes unification…

- Can be source 2 source, source 2 bin, bin 2 bin, …

Obfuscation & research

- Obfuscation is a very secretive area
  - Pretty much no publications, some very obscure patents
  - You have to sign 20 NDAs when working on attacking obfuscation
    - Players: Google, MS, Adobe, Apple vs. Arxan, Cloakware, Safenet, …
    - Also some home-made obfuscation (remember the “dont do crypto if you are not a cryptograph” ?)
- I can just tell that we broke the crypto in most of the DRM we attacked

Don't solve the wrong problem

- Problem #1: analyzis
  - Exactly the same constraints as for malware or vuln research, no need of a specific focus
- Problem #2: “difficult”
  - Define what is “difficult” to analyze…
  - Assessment always performed on the binary

Challenge: get a complexity measure

- Source or binary analyzis keeps saying “program is difficult to analyze”: use that as a metric for obfuscation?

- Metrics on the CFG, the diversity / similarity in a code (to avoid pattern matching)

- What about the data flow?

  - Today: MBA or float are used because no tool supports them properly

Obfuscation REALLY needs to have a good measure for complexity

Bonus: obfu as crypto

- Apply Kerckhoffs principe to obfuscation’s design

We need a strong obfuscation that can “resist” reverse while every detail is public

Crazy little thing called patch

Patching as a challenge

- Diffing code is a well-known way to find vuln
- Diffing binaries has been used a lot to find vuln
- But how to patch once a vuln is known?
- Even better: how to patch an unknown vuln?

When patching makes a difference

- Desktop: most are relying on automatic updates
- Mobile: automatic updates will come there too
  - iOS does it already, Android will anytime soon
- Embedded | critical systems: …
  - IoT: 2 billions devices to patch at once?
  - Critical systems: some can reboot only once a year
- Abandonware: yes, it does exist also for important systems

Dynamic patching: memory access

- White list: every access is forbidden, it is granted only once it is proven to be safe
  - Very generic, can patch even unknown bugs
- Black list: every access is granted except where a bug was found (e.g. by fuzzing) and a bit around
  - Bug specific but avoid to analyze the root cause

Dynamic patching: input validation

- Determine the list of inputs for a program
  - From network, files, user, …
- Wrap all the inputs with a function call before initialisation
  - Can use white or black list approaches

One patch to rule them all

- Applying the patch
  - Statically: modifying the binary :/
  - Dynamic: inject in the process, load a library, …

Additional challenges

- Performance: the patch must have a minimal impact on the target program
- Soundness: the program must work in the same way (with at least a bug removed): can we “prove” it?
  - Re-run with the bug triggering sample
  - Try with many other samples

QUEEN

THE SHOW MUST GO ON

Conclusion

- There are still a lot of challenges
- Please, think big first, then sound
- TOP 3 :
    - Similarities based on many big files
    - Make code analyzis usable
    - Patching unknown vuln
- Bonus: think different: firmware, complete system, bootloader…

Questions?

Fred “pappy” Raynal
fraynal@quarkslab.com
@fredraynal