LINUX* KERNEL CODE REVIEW

Mark Gross – Instructor
Customized for ELC2017
Course Code - NNNNNNNN
INTRODUCTION

- This is a talk derived from a class I have taught at Intel.
- It is from the perspective of a vendor or integration tree provider with OEM customers.
- This talk is influenced by experience delivering Linux* kernels (aka evil vendor trees) into Android* stacks that go to customers.

*Other names and brands may be claimed as the property of others.
The examples are a bit tame. I wanted to limit my examples to publicly accessible code.
COURSE GOALS

• At the end of this class you will understand what code reviewers are expected to look for as they review changes going into a product specific Linux® Kernel.
  o With this understanding you will know what is expected from your own code in a code review.
  o You will understand the mindset of customers.
  o You will understand the utility of good prefix discipline.
  o Identify a problem patch that needs to be fixed.
  o Be able to explain what is the issue with the patch

*Other names and brands may be claimed as the property of others.
PREWORK

• Did you complete your reading?
  o How to Get Your Change Into the Linux Kernel
    http://users.sosdg.org/~qiyong/lxr/source/Documentation/SubmittingPatches
  o Linux Kernel coding style
    http://users.sosdg.org/~qiyong/lxr/source/Documentation/CodingStyle
WHAT CUSTOMERS THINK OF YOU AND YOUR CODE.

- They do not trust you or your code.
- They do not want changes after “beta” releases other than bug fixes they care about.
- They scrub *every* change in the tree in detail.
- They do their work on initial releases provided and want to bill you for the overhead associated with the refresh of their trees after every update you ship.
WHAT I THINK ABOUT CUSTOMER KERNEL PRACTICES

They are System Integration and Sys-debug teams.

- They often can't understand using anything but Gerrit to manage technical debt.
- They don't plan for security maintenance.
- They don't plan for rebases or migrations.
- They don't upstream bug fixes they depend on.
- They don't provide visibility on changes they are making.
- They are driven by TTM, risk and short term costs.
HIGH LEVEL POLICIES

• Patches MUST follow:

• As a reviewer its important to enforce this policy.
COMMITEE PREFIX CONVENTIONS

"UPSTREAM:" The commit comes from upstream from a later kernel version, and its original SHA is noted in a “cherry-picked from ...” line at the end of the body of the commit message, before the committer’s Signed-off-by line.

"BACKPORT:" The commit is an “UPSTREAM” commit, as noted above, just had conflicts that needed to be resolved. The conflicts are noted at the end of the body of the commit message.

"FROMLIST:" The commit is from an upstream mailing list, and is likely to be accepted into upstream, but has not yet landed. The mailing list archive URL to the commit, or Message-ID, is noted at the end of the body of the commit message.

"ANDROID;, BRILLO;, CHROME;, YOCTO;: ... " The commit originates from the respective OS specific kernel tree, and is not yet upstream.

"VENDOR: vendor-name:" The commit comes from a vendor (where “vendor-name” is replaced with the vendor), and contains commits not yet upstream.

"RFC:" A temporary state where comments are requested before attempting to upstream the commit (after which it would move to “FROMLIST;”)
Bad patch examples:

Git show 74c11ba
Git show 4ce179e

Picture a tree with 500 to 5000 patches in a single branch gerrit tree with up to 50% of the patches being like this randomly distributed and then needing to do kernel migration.

Say, for a new SKU in your company's product line needing an extended shelf life in a security sensitive application.
Can someone, capable of reading C-code and knowing a little about kernel programming, make progress debugging issues with any of the patches without calling for support?
SUPPORTABILITY

• Commit comments shall be meaningful and accurate
  o Explains what will break without the change.
  o Explains why the change is important to take.
  o Explain what the change is.
  o Match the actual change.
SUPPORTABILITY

• Good commits are sentences in the story of the tree history.
  o They each need to stand alone and make sense just like complete sentences.
  o Sentence fragments are bad. For example, when a header is changed to add a define but nowhere in the patch is the define ever used.

Many times incomplete or fragment commits result form porting patches forward.
Delay loops are a problematic use of magic numbers and need special attention. Every log message is a potential support call that costs money. We need to make them count!
Nobody likes being forced between accepting a regression and a needed feature. Don’t be that guy.
**SUPPORTABILITY**

- Imagine yourself one year from now in a hot customer escalation where something points at this patch under review as 'guilty'.
- or
- Imagine 2 years from now when you can no longer provide effective security maintenance and are forced by business needs to migrate the patch to a new kernel baseline.

- Is the description/code/comments clear enough to then figure out what this patch is about?
- If needed, can I then keep the goodness in it even if I need to redo some of it?
EXAMPLE HERE

Positive example of a good quality patch that is hard to say no too even on a bad day.

Git show 6d1cc7ba
Does it explain what will break if the patch is not accepted? alarmtimer_suspend  CONFIG_RTC_CLASS example.
CORRECTNESS

• Is memory allocation and deallocation done properly?
  o Check error paths for memory leaks.

• Check global variables.
  o Check / ask if the global is really necessary.
  o Check / ask if the global needs an associated lock to protect it from concurrent access.
CORRECTNESS

- Is the locking sane? Is the locking model documented?
  - Remember locks protect “data” (not code) from concurrent access.
  - Is what is being protected by a lock documented?

- Review the static analysis output and use good judgment:
  - Is the code conforming to coding standards? (Remember the reading from the prework!)
  - Is the code adding compiler warnings?
  - Is the code passing static analysis checks?
A WORD ABOUT CHECKPATCH.PL

- We cannot treat checkpatch as a judge.
- Checkpatch.pl will sometimes call out errors that cannot be fixed and as such we can’t rely on it within an automation setting.
- We need kernel reviewers to review the checkpatch output and decide if it needs to be fixed.
- Developers need to be ready for challenges from the reviewers WRT checkpatch warnings and errors.
CORRECTNESS

• Security issue scanning:
  o Review the security analysis results and be very careful before accepting the patch if there is any issue identified by the scan.
  o Ask for help if you are not sure before dispositioning a possible security issue as false positive.
CORRECTNESS

• Is the code compliant with IP plans and guidelines?
• Is the code in compliance with Legal and business policies:
  o Does the patch leak confidential information in the comments?
  o Does the patch touch un-documented registers or use previously undocumented values in
    MSRs?
  o Business approvals exist for the IP the code is for.
  o Where was the content sourced from (see next slide)?

 o Does it leak new IP information by its implementation?
 o Is that IP information approved for publishing under GPLv2?
- Make sure proper attribution is provided to code adapted from any external source.
  - It's more than just a plagiarism issue.
  - Note knowing how to use "git --amend --author=" is an acceptable excuse.
  - Customers need to know the background for our patches so they can have more trust in them.
  - If you are pulling ideas/code from external sources you have to point it out and provide attribution.

- It’s an easy mistake to do when you are in a hurry.
- Providing background on patch origins provides credibility via herd mentality.
- It’s important to have a good perception of yourself and your employer in the open source community.
Help rewrite problematic commit comments or code if needed.
Details of a Good Code Review

- Provide actionable feedback to the author.
  - Making clear what feedback must be acted on to get merged.
  - Provide explanation for why the code needs to be changed.
  - Try to make sure all the issues are reported in the first review. We want to avoid harassing the author with multiple review cycles where new existing issues are raised after the author addresses whatever was commented on in the earlier reviews.

- Do not compromise on supportability! It is likely to bite you and your company in the future if you do, with customer escalations.

Note: mistakes happen but, we want to avoid a “bring me a rock” situation.
Many companies have experts you can ask for help from. Use them when you are unsure as a reviewer.

Take notes and learn from the experience! Don’t miss out on the opportunity for you to grow yourself technically!
CALL TO ACTION / KEY TAKE A WAYS

Understand how patches are used by customers.

Review patches assuming they need to be reused across kernels and your business depends on efficient reuse.

Use good prefix conventions!

Stop writing bad patches with crappy commit comments:

- When you write bad patches nobody likes you.
THANK YOU! --- QUESTIONS?
Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation