How to Handle Security Flaws in a Storage Product Using Open Source Code

Jeremy Allison / Google / Samba Team
All new storage products use Open Source

• Economics drive this.

- Underlying OS is Linux (usually) or FreeBSD.

• Unless you employ Linus or other notable names, you don’t have full control over what goes into your product.

• You **must** have a process to coordinate with Open Source upstream developers in order to ship secure products.

- At the very least, you need to know about vulnerabilities in the code you’re using, even if you don’t (or can’t) fix it yourself.
Dealing with upstream vulnerabilities

• Ensure the upstream project takes security seriously.
  
  – This is not as common as you might think – do you have a contact point if someone reports a security flaw to you?


• Even projects that do security well themselves have dependencies.

  – Know what is going into your storage solution.
A story of three (Samba) flaws

• “Badlock” and industry-wide coordination.

- “Trust no one” (with apologies to the X-files).

• Sambacry.

- “Anything you can do, I can do better..”

• Google Project Zero bug.

- Practicing for the real thing.
• The first security flaw reported in Samba (1993) was immediately caught by Andrew Tridgell (tridge) – the original author of the project.

  - He stopped the mail list processing until he had a fix.

  - Ensured the very next email contained the patch.

  - Re-started mail list processing.

• Things are a little more difficult these days.
• Put a process in place to handle all security reports uniformly.

  - Start with an email alias: security@samba.org

  - Can be hard to do with a pure volunteer organization, but without it you’re not professional.

• Ability to get Common Vulnerability and Exposure (CVE) number is essential for tracking.

  - Linux distributions are your friends here, their security Teams can handle this for you.
Auditing / Code quality?

• Unless the Open Source project is large and important, no one will audit it for free.

– Automated tools for static analysis and fuzzing are essential.

– A comprehensive test suite helps automate the testing needed.

• Basic code reviews from people with security experience will help catch the worst errors.

– If you don’t have security experience, shipping code will soon teach you :-).
The reputation game

• Use gpg encrypted email to communicate with vulnerability reporters.
  – Standard in the security world.

• Insist on transparency with security researchers and in vulnerability disclosure.
  – Don’t try and hide anything – you’re not fooling anyone.

  – Ignore vulnerability-sellers.

• Internal and external time-frames can differ, but try and stick to a schedule.
How to respond

• Insist on reproducible exploit to fully understand the threat.
  – You don’t have to publish these!

• Don’t race for the “easy” fix.
  – Take time, understand the issue and look for it in all areas of the code.

• Only fix the security bug.
  – Don’t try and fold in other bug fixes for a security release.

• Limit back-ports / Coordinate with vendors.
Case study #1 – Bad, bad, badlock

• "Badlock" was a protocol-level vulnerability in DCE-RPC (remote procedure call), used by all Microsoft interoperating products.

– Complex, and almost no one understood it (except exploiters, who might have already been using it).

• Discovered indirectly during a Microsoft Interop Event by a proprietary fuzzing tool.

• Tension occurred between commercial interests of employer of discovering engineer and Samba project (my fault).

– Don’t let marketing people name bugs :-).
“Badlock” affected most SMB implementors, so coordination had to be arranged across the entire storage industry.

- Knowledge of the bug started to leak.

- Attacks on Samba bugzilla by black-hats attempting to get early advantage.

- Personal contacts essential (reputation again). I started refusing to discuss unless I personally recognized the phone number/voice.

- Seven months from discovery to coordinated released fixes. “90-day” window would have killed us here.
Badlock postmortem

• Most of the press completely failed to understand or report on the threat correctly.

– Most security “researchers” completely failed to understand or report on the threat correctly.

• Worst-case scenario – thankless fix misunderstood by users and anyone not intimately involved in the code.

– Hard to get management support.

• Don’t try and create catchy names and logos for bugs.
• Tod Beardsley (security researcher at Rapid7) tweeted:

“Microsoft SMB: Wow, what a week! Samba: Hold my beer”
Case study #2 – Sambacry

- Caused when two secure subsystems - module loading and named pipe services - were connected without sufficient input checking.

  - Code was in error for seven years.

  - Externally reported.

  - Unknown how much it had been exploited.

- Fix was a one-line change.
Sambacry postmortem

- Better security review would have caught this.

  - Impossible to catch everything.

  - Logic error, not language error (safer language would not have helped).

- Tests both positive and negative would not have helped, they would only have showed the named pipe module loading worked or failed.

- Worst effect was non-upgradable embedded systems with old unfixable versions.
Case study #3 – Google Project Zero

• Project Zero Google security researcher Jann Horn (he of the “Meltdown” and “Spectre” attacks) cut his teeth on a Samba bug.

– Even though I’m a Google employee, we didn’t get any slack :-).

• “Borderline” exploit – race condition in pathname processing (required slowing the server down with strace in order to hit the race).

• Exposed generic design flaw in user-space server code.

– Goodness knows how or even if other servers have fixed this.
Google Project Zero mitigation

• Required redesign of all pathname processing.

  “Natural” way to fix this turned out to be covered by a software patent.

  Thankfully a superior solution was not covered by patents.

• Immediate fix took around one week.

  Then we discovered the fix broke one of the critical VFS modules.

  Module was created for the needs of the patent holder covering the original solution :-).
Google Project Zero mitigation

• Ultimately took the full 90-day disclosure time, plus a 14-day extension, to get the fix created, tested and back-ported to all vulnerable versions.

– Security work under time-pressure is when mistakes happen.

• I am ambivalent on deadlines, they ensure concentrated effort but can do harm.

• Ensure you explore all combinations of design decisions for robustness (I know, this is impossible :-).

– Code fail-safe. Just because “it can’t happen” doesn’t mean someone won’t find a way to do it.
Google Project Zero postmortem

• Design flaws are the hardest problems to fix.
• Don’t try and argue / push back on vulnerabilities with security researchers.
  – Even if you’re convinced you’re right, when they go public it will still damage your project reputation.
  – Work with them to agree on a mitigation strategy.
  – Don’t be embarrassed to beg and grovel to get more time.
A thankless task

• No one rates security until they don’t have it. Even then, not so much.
• The press **WILL** completely mess up all reporting – security flaws are complex even for exports.

  “A flaw in Microsoft’s implementation of the Samba protocol…”

• Volunteer developers will get blamed and called fools.
• Personal contacts are essential for coordinating fixes.
• Security work is like ensuring the sewers stay open.

  - No one notices until you fail.
Conclusion

• Prepare for massive overwhelming security failures in your project.

  – That way, when it happens (and it WILL happen) at least you have a plan.

• Accept all reports, respond to all reports.

  – Even if they appear insane.

• “Untested code is broken code”

• There is no magic bullet / magic language that will protect you.
Questions and Comments?

• jra@samba.org
• jra@google.com