

How secure code review is different than exploit development



This page has been made public for vendors

Question

How is secure code review is different than exploit development?

Answer

The goal of secure code review is to make code as obviously secure as possible[1]:

Steer Clear of the Exploitability Trap

Security review should not be about creating flashy exploits, but all too often, review teams get pulled down into exploit development. To understand why, consider the three possible verdicts that a piece of code might receive during a security review:

- Obviously exploitable
- Ambiguous
- Obviously secure

No clear dividing line exists between these cases; they form a spectrum. The endpoints on the spectrum are less trouble than the middle; obviously exploitable code needs to be fixed, and obviously secure code can be left alone. The middle case, ambiguous code, is the difficult one. Code might be ambiguous because its logic is hard to follow, because it's difficult to determine the cases in which the code will be called, or because it's hard to see how an attacker might be able to take advantage of the problem.

The danger lies in the way reviewers treat the ambiguous code. If the onus is on the reviewer to prove that a piece of code is exploitable before it will be fixed, the reviewer will eventually make a mistake and overlook an exploitable bug. When a programmer says, "I won't fix that unless you can prove it's exploitable," you're looking at the exploitability trap. [...]

The exploitability trap is dangerous for two reasons. First, developing exploits is time consuming. The time you put into developing an exploit would almost always be better spent looking for more problems. Second, developing exploits is a skill unto itself. What happens if you can't develop an exploit? Does it mean the defect is not exploitable, or that you simply don't know the right set of tricks for exploiting it?

Don't fall into the exploitability trap: Get the bugs fixed!

If a piece of code isn't obviously secure, make it obviously secure. Sometimes this approach leads to a redundant safety check. Sometimes it leads to a comment that provides a verifiable way to determine that the code is okay. And sometimes it plugs an exploitable hole. Programmers aren't always wild about the idea of changing a piece of code when no error can be demonstrated because any change brings with it the possibility of introducing a new bug. But the alternative—shipping vulnerabilities—is even less attractive.

Beyond the risk that an overlooked bug might eventually lead to a new exploit is the possibility that the bug might not even need to be exploitable to cause damage to a company's reputation. For example, a "security researcher" who finds a new buffer overflow might be able to garner fame and glory by publishing the details, even if it is not possible to build an attack around the bug [Wheeler, 2005]. Software companies sometimes find themselves issuing security patches even though all indications are that a defect isn't exploitable. [...]

References

[1] Brian Chess and Jacob West, 2007. Secure Programming with Static Analysis. Chapter 3 Static Analysis as Part of the Code Review Process (Free Sample Chapter). Addison-Wesley Professional. Retrieved from [this](#) location.