The Wassenaar Arrangement's intent fallacy

Sergey Bratus

December 8, 2015

The current definition of "intrusion software" aimed to provide a description of a class of technological capabilities that is independent of their intent. The idea to judge a technology based on its technological intrinsics alone rather than on its uses is appealing, as it avoids the complexity of judging intent. This approach has established itself in export control regulation—but, unfortunately, it fails for software in general and for the Wassenaar's goals in particular. Although WA's language may seem technical, it is in fact *pseudo-technical* and suffers from a fundamental misunderstanding of technological issues. This note explains why.

Specifically, WA's attempt to define an intrinsic technological capability as *"The modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions."* fails dramatically. This language would make sense if "standard execution path" were somehow a property of the code that could be unambiguously gleaned or derived from it—but it can't and it isn't.

Instead of removing the element of intent, this definition, in fact, is implicitly but entirely dependent on the intent of developers of the vulnerable software. Thus, the WA definition fails to accomplish exactly what it aimed to do: describe a technical capability in an intent-neutral way.

The word "standard" in the definition masks this dependency on developer intent, but it too fails. Indeed, in computing, "standard" means a description of how software or a device *should* or *must* operate, i.e., is intended to operate. There is no "standard" execution of software, because there is nothing to set such a standard other than the software's specification—which describes its desired, expected, intended execution.

Describing a "standard" or intended execution as "execution that happens most of the time" fails, because it does not account for handling of special conditions that arise rarely, but still must be handled correctly. Furthermore, its does not account for deliberately added special features that may be called upon only rarely but are still implemented for backward compatibility. Such "mostly-forgotten" features may be used by either benign users or attackers; in fact, they served as attack vectors for several recent high-profile vulnerabilities. It's only in hindsight that the developers could and did say, "No, no, we did not mean it that way!"—in some cases, over 20 years later.[1]

Unfortunately, capturing intended operation of software is far from a solved problem in computer science. Simply put, there is, in general, no technical way to find out what a program is supposed to do or whether it will actually do it. We can ask the programmers about the original intent of their programs, but, in general, we cannot even verify if they are honestly mistaken about their program's actual behaviors, let alone derive their intent from the code.[2]

Indeed, if we had a way to unambiguously express and verify programmer intent, or were able to unambiguously tell "normal" executions from "abnormal" ones, spotting vulnerabilities would be simple, cyber-attacks would be easily detected and mitigated, and WA itself would not be needed.

[1] Cf., e.g., the *Shellshock* vulnerability.

[2] The popular programmer joke "It's not a bug, it's a feature" refers to the difficulty of distinguishing between some cases of erroneous program behavior and intended behavior, and has more than a grain of truth in it—some argue that it is in fact truth with a grain of joke. Such ambiguity may seem to be rare, but it frequently occurs with security vulnerabilities—because this ambiguity plays a prominent role in creating them.

Designers, vendors, and programmers themselves have trouble describing whether a software or hardware feature is operating as intended or is, in fact, a security flaw. Advanced exploitation techniques are rapidly moving towards *"the boundary between bug and expected behavior"*, *"almost the fringe of what can be classified as an explicit hole or flaw."*[3] Advanced exploitation is rapidly becoming synonymous with the system operating exactly as designed—and yet getting manipulated by attackers.

Entire active research disciplines are dedicated to producing more accurate descriptions of programmer's intent (including type theory and formal semantics). Invention of new programming languages is driven by the need to better capture programmer's intent to make programs more reliable. Practical programs that can be automatically and unambiguously verified to correspond to their intents to date require very special ways of programming from the ground up; only a handful of such programs exists.

Then take "modification". Modification of execution paths is the essence of software composition, a key computer engineering phenomenon, and a deep research topic. Every debugger, every dynamic linker, every OS trap handler, every virtualization primitive performs it, whether or not intended by the original programmers. Patching, an important security tool, is nothing but modification of the execution paths to include the externally provided instructions of the patch.

The WA itself exempts some of these modification categories explicitly—but, of course, cannot and doesn't cover all kinds of software composition and instrumentation. In short, the Wassenaar language here is ignoring the very basics of systems and software engineering.

Finally, the additional qualifications that WA places on "intrusion software", such as bypassing *"protective countermeasures"*, fail to meaningfully narrow it down—as those exploitation techniques that don't are already mitigated and are thus of little practical value to the industry defenders; researchers consider them obsolete for the same reason. Moreover, here WA also seems to confuse intent with reality, implicitly assuming that protective measures unequivocally function as intended, to benefit the defenders. In reality, such measures often obstruct fixing weaknesses, and are also used by malware such as ransom-ware to protect itself and hinder recovery of the infected system—and thus methods to bypass them must be routinely developed by the defenders.

In summary, WA's definition, despite its technical-sounding jargon, is no better than saying that "intrusion software" is whatever makes a program do what its programmers did not mean it to do, somehow.[4] This brings human intent back to what was supposed to be a pure technological capability description—in a way that is ambiguous and given to after-the-fact interpretation by vendors and programmers.

[3] Aaron Adams, "Research Insights: Exploitation Advancements", NCC Group, 2015

[4] This, naturally, captures most defender fixes, as they necessarily change the program's behavior, and must be applied even if the program was not originally designed to accept them.