Why Wassenaar Arrangement's Definitions of "Intrusion Software" and "Controlled Items" Put Security Research and Defense At Risk

Sergey Bratus, Michael Locasto, Anna Shubina

July 23, 2014

1 Definition of "intrusion software" and "controlled items" in Wassenaar Arrangement

Wassenaar Arrangement (WA) uses a two-step conceptual structure to capture the kinds of surveillance-related software it purports to control. First, it introduces the concept of *"intrusion software"*, which it defines as

> Software specially designed or modified to avoid detection by 'monitoring tools', or to defeat 'protective countermeasures', of a computer or network capable device, and performing any of the following:
>
> a. The extraction of data or information, from a computer or network capable device, or the modification of system or user data; or
>
> b. The modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions.

This new class of software so defined is very broad and fundamental. As we will show it not only covers software known in computer security jargon as *exploits* and *rootkits*, but in fact all elementary means of software instrumentation, construction, and deconstruction outside of the software's pre-defined interfaces for these purposes—despite explicitly excepting "hypervisors, debuggers or Software Reverse Engineering (SRE) tools" in the note to the above definition.

However, WA does not directly control this new class. Instead, it defines a *second*—controlled—class of software and systems derived from the "intrusion software" class, namely those associated with "generation, operation or delivery of, or communication with, 'intrusion software"' and those for its "development" and "production." In particular, subjected to control are:

> 4. A. 5. Systems, equipment, and components therefor, specially designed or modified for the generation, operation or delivery of, or communication with, "intrusion software".
>
> 4. D. 4. "Software" specially designed or modified for the generation, operation or delivery of, or communication with, "intrusion software".
>
> 4. E. 1. c "Technology" for the "development" of "intrusion software".
>
> "Software" specially designed or modified for the "development" or "production" of equipment or "software" specified by 4.A. or 4.D.
>
> "Technology" according to the General Technology Note, for the "development", "production" or "use" of equipment or "software" specified by 4.A. or 4.D.

The apparent rationale for this two-step definition is that attempting to control elements of malware as such would inhibit communication between malware researchers and discovery of new vulnerabilities (a judgment we agree with), while controlling the second class derived from the first would limit WA's control to means of developing and delivering malware.[1]

[1] We take this explanation from `https://www.privacyinternational.org/blog/export-controls-and-the-implications-for-security-resea update`

Unfortunately, this two-step structure causes more problems than it solves. By starting with an overbroad definition of the "intrusion software" class that serves as a basis for the second definition, it subjects to control the primary known means through which research and engineering progress has been made in all known aspects of software, including security. These means, as we show in Section 4, are automation of generation and operation of software elements.

If we recognize that the first class of software, despite containing potential building blocks of malware, must not be chilled for the same of research and deepening our understanding of software flaws, then we must recognize that the very means by which it is advanced—automation of generation and operation—should not be chilled either. Moreover, chilling these means will cause more harm and more power imbalance in the long run, because freezing engineering progress of software undoubtedly favors existing power players and protects them from disruptive technologies developed by small private parties.

We first discuss why the (uncontrolled) "intrusion software" class is too broad and why exceptions written into it don't help. We then look at why definition the controlled class of technologies based on the present definition runs against the fundamental trend of engineering progress, and is therefore an ever bigger danger.

2 Why "intrusion software" definition has wrong granularity that exceptions cannot fix

Wassenaar Arrangement defines "intrusion software" (and thus also "controlled items" for "generation, operation or delivery of, or communication with" or "development" of "intrusion software") in terms of fundamental *operations* of computer science research and software engineering (generally speaking, as fundamental as operations such as taking roots, proof-by-contradiction, or variable substitution are to mathematics). These operations are present in all non-trivial innovative software (see Section 5) and are critical for pushing state-of-the-art security research (and not only security research) forward; they are especially critical for improving defense. At the same time, exceptions to these definitions ("Hypervisors, debuggers or Software Reverse Engineering (SRE) tools; ...") are at a different, much higher level of *whole programs* or products built for a particular purpose.

Complex software is built in multiple levels of aggregation and composition. Innovation entails aggregation and composition in unforeseen combinations, at many levels. Thus WA essentially whitelists some combinations and compositions (those that are seen as important to software engineering *today*) but not the myriad of others (which will become equally or more important tomorrow).

The excepted programs or products contain both components with functionality labeled "intrusive" and other components. For example, a debugger contains software for "modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions"—such as a module that injects breakpoints to divert control of the debugged program or device—and a GUI. Without a GUI, the breakpointing software component could be considered "intrusion software"; a reasonable judge who had even practiced debugging with an integrated production debugger may be swayed by the argument that software that lacks a GUI is *not* a debugger and thus not excepted.

Yet it is in these execution-modifying (or "execution-hijacking") components that progress in debugging tools—and thus in observability and, in turn, in security of software—is made. For example, Microsoft's release of its Detours library was a significant step forward. Detours' description by Microsoft fits both "intrusion software" and "controlled items" WA language.

> "Detours intercepts Win32 functions by re-writing the in-memory code for target functions. The Detours package also contains utilities to attach arbitrary DLLs and data segments (called payloads) to any Win32 binary." —`http://research.microsoft.com/en-us/projects/detours/`

Detours can be used as a component of a debugger or as a part of malware. For example,

> Malware authors like Detours, too, and they use the Detours library to perform import table modification [standard technique of diverting standard execution paths into new code/commands supplied by a Detours user], attach DLLs to existing program files, and add function hooks to running processes.

> Malware authors most commonly use Detours to add new DLLs [containing their malicious code/commands] to existing binaries on disk. [discussion of various malware authors' techniques follows]
>
> — *Practical Malware Analysis*, Michael Sikorski and Andrew Honig, No Starch Press, 2012, page 262, Chapter 12.

Detours implements the pattern of modifying the execution path of other programs known as "hooking". Hooking is a basic pattern that is used ubiquitously for all kinds of software composition. It is used for debugging, instrumentation, and performance tuning of software, as well as for patching vulnerabilities in it, and for upgrading software that has been shipped with no dedicated upgrade mechanism (an important security capability for legacy software, such as software that runs critical physical infrastructure).

Moreover, Detours is popular with developers, because

> the Detours library makes it possible for a developer to make application modifications simply.
>
> —Ibid.

The hooking pattern implements the fundamental software engineering operation of *composing* software with other software. For this reason, it is implemented by a great variety of software, with a wide range of techniques and uses. Often the hooking software is developed and released separately; sometimes it is also released together with management tools and automation tools.

Detours also includes a *management* component for its means of modifying the execution path, and for *automating* the actions that deploy these means. These components would fit the WA definition of controlled items, since they help operate, manage, and automate application of Detours' means of execution path modifications.

2.1 Bypassing "protective countermeasures".

Wassenaar language targets "defeat[ing] protective countermeasures", explained in a footnote as "techniques designed to ensure the safe execution of code, such as Data Execution Prevention (DEP), Address Space Layout Randomisation (ASLR) or sandboxing." But what does "defeating" mean? This language appears to include any software composition (such as patching or jailbreaking) that work reliably on systems with a "protective countermeasure" enabled.

ASLR. For example, the point of ASLR is to make the location of various software components when loaded into the memory of a computer less predictable. To patch such software while it's running (such patching is known as *hot-patching*, used for servers and other devices, including mission-critical devices that are expected to operate 24/7), the patching software typically scans the computer's memory and identifies the addresses (locations) that were "randomized".

This memory scanning technique is one of the most fundamental research and engineering operations. Software that performs this non-trivial operation and looks for patterns in memory can be developed and distributed on its own, or with other components such as pattern-matchers for memory contents or memory visualizers. In any case, it can be said to "defeat" ASLR, by making available to its operator the information obscured by ASLR; a reasonable judge familiar with technology would find this statement true at face value.

For example, the F.L.I.R.T. technology is used by IDA Pro, a reverse engineering tool to locate library functions, which are obscured by ASLR. F.L.I.R.T. is identified by the tool's maker as a separate technology.[2] Since its publication, other software based on the same principles and dedicated to the task of scanning memory at runtime has been developed by various parties and has aided in a variety of applications such as forensics and hot-patching.

The operation of scanning memory to locate specific software components is too fundamental and low-level to ascribe to it any intent or any specific use; yet it "defeats" the obfuscation imposed by ASLR by definition.

[2] `https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml`

Sandboxing. Sandboxing is a key engineering practice of limiting a program or a device in its access to system resources. However, since the engineering practice is so generic and ubiquitously used, bypassing the restrictions of a sandbox is also frequently used.

For example, *jailbreaking* of mobile phones to bypass manufacturer restrictions "defeats" sandboxing. To make it easy for non-technical users to jailbreak and "unlock" their iPhones, developers of the jailbreak delivered the jailbreaking commands through a browser exploit (altering the execution path of the browser software); the user merely had to navigate to a webpage to get the jailbreak take effect (delivered).

All of these activities, including those allowing users to customize and protect their phones, would be covered by WA language.

3 Automated Exploit Generation

WA control lists specifically target "generation" and "development" of "intrusion software". Thus they apply directly to generation of exploits, which are means of modifying the execution path of software.

Automatic generation of exploits is a rapidly developing direction of security research. The promise of this field is to identify and prioritize security-critical software bugs so that they can be eliminated. Prioritization is important, because modern complex software contains a multitude of bugs, many of which are not exploitable; demonstrating that a bug is exploitable generates the exploit, which scientifically and incontrovertibly proves this fact. In the words of the leading academic group that coined the term AEG,

> The generated exploits unambiguously demonstrate a bug is security- critical. Successful AEG solutions provide concrete, actionable information to help developers decide which bugs to fix first.

Although the name "Automatic Exploit Generation" (AEG) does not suggest it, AEG is in fact a task closely connected with *software verification*, a research and engineering methodology that uses formal methods to secure software. Continuing the above quote:

> Our research team and others cast AEG as a program-verification task but with a twist [..]. Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an "exploitability" property, and the "verification" process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a *firm theoretic foundation*. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical.
>
> —*Automatic Exploit Generation*, by Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley, Communications of the ACM, February 2014, Vol. 57, No. 2, p. 74

In a nutshell, AEG is a promising method of containing vulnerabilities that is based on firm theoretic foundation of proven computer science.

As with fuzzers, development of industry-strength AEG engines starts with prototypes built by individual researchers or academic groups, but then moves to commercial startups to accommodate the scalability, performance, and other engineering challenges that require dedicated effort of professional developers. Yet this is also the stage in which such research produces its most fundamental results and proves its ability to handle real-world software. Chilling AEG would severely set back defense.

4 Why WA control items will impede progress of software security

We referred earlier to the apparent rationale for the WA language not controlling so-called exploits or rootkits, but instead controlling the software that is used to "generate" or "operate" or "deliver" exploits, and to develop all the above.

Several points must be made about this language:

1. It presents fundamental obstacles to engineering progress of security tools, and to offensive research in particular.

2. Its practical application to actual research and engineering artifacts used in offensive research is just as vague as that of "intrusion software" or potentially even more vague.

This language presumes a clear boundary between programs that implement a particular software functionality and the programs used to create such implementations. In reality, no such clear boundary exists.

The structure of classifying software and the way that software progresses is misconstrued in the underlying concepts of the supposed dichotomy. *In fact, all of our technical examples above easily fall into the controlled category of "intrusion software" enablers!*

Progress in software engineering is being made by abstracting functionality from products first into libraries and then into domain-specific languages and development tools. Early computers took a single program (modern low-end microcontrollers still do), later computers required a specialized program to *operate* other programs; this program is now known as an *operating system*. Early programs were written in the basic commands of the computer, and realized basic conceptual elements of programming such as loops and conditionals in these basic commands; later programs were written directly in terms of these conceptual elements, and required specialized programs to *generate* the actual basic commands or to emulate them. These specialized generating programs became respectively known as compilers, interpreters. A middle ground was taken up by "virtual machine" programs, such that run automatically generated hybrid "bytecode" commands for Java and .Net programs and at the same time "operate" them.[3]

Thus parts of functionality continually move from *programs* to the *libraries* (which standardize both operation and programming) and the "tools", and specifically by way of tools "learning" how to generate what used to be code in the main program body.

Without this migration of logic from programs to "development tools", without thus abstracting away the complexity, progress in programs is impossible. But under WA logic, this migration would create controlled items even if the programs themselves are not controlled. Thus *abstraction*, the key means of deepening our understanding of both engineering and research issues, will be chilled.

When does code for some functionality stop being a part of an uncontrolled program and becomes a controlled "tool"? Does this happen when it moves to a library? A shared library? A piece of environment that must be present for the main program to operate? When it moves into a tool to be generated automatically from an abbreviated instruction or statement or code line in the program?

Moreover, not every code that is automatically generated is generated by a compiler. It may be generated by several levels of scripts from templates, by a *Makefile*, or a scripted build, by any part of the build system, and so on. Present day's build systems are complex and multi-layered, and each layer creates automatically generated code. There are no clear boundaries where code templates end and "generated" code begins.

It is only thanks to this progression of automating operation and generation of programs that we were able to advance from relatively small and simple programs to the present state of software engineering and research.

Security research follows the general pattern of software engineering. There is broad recognition among security researchers that the better, more principled kind of defenses that common operating systems employ now, commercially known as DEP, ASLR, EMET, and others are a result of *co-evolution* of offensive research and defensive systems research.

Advancement of offensive research, key to this co-evolution, required substantial engineering investment—into exactly the kind of "generation" and "operation" aspects of offensive software. In full accord with the general trend described above, so-called exploits and rootkits went from entirely hand-coded for the occasion to use of libraries, then to specialized compilers, build systems, interpreters, and remote proxying designs comparable with production virtual machines emulators.

For example, initial defenses against the Return-Oriented Programming techniques (so known since the academic publications of 2007-8, but known to offensive researchers since at least 1999-2000) did not take into account the fact that finding of the snippets of code in the target that were composed by the attacker to program the target without introducing any binary code could be automated. While it was clear to security

[3]Such are the Java VMs inside web browsers and inside Android phones.

researchers experienced in offense that automation was possible and likely, and also that a skilled attacker would need far less than complete automation to bypass existing defenses, the threat was not so clear to vendors.

It took building actual *ROP compilers* software to perform these tasks automatically and in a platform-independent manner to present the defenders with a proper yardstick for testing their actual and proposed system defenses. Yet ROP compilers clearly fall among the WA controlled items.

FILL Fuzzers?

Automating operation and generation of code is the only way of making progress. Operational automation and generation of code by tools is how software engineering makes progress. They enable us to write larger programs, but that is less than half of the story—they also enable us to see what actual challenges and possibilities come to the forefront when we reach each level of scale and complexity.

It used to be that the job of system administrators was to manually enter commands to operate systems in their care. Automation of these commands in common operational scenarios was what made Cloud Computing possible (while dropping costs of hardware made it economically feasible in its present form). Automation is at the core of every engineering advance; in computing, it is generation of logical commands or code that gets primarily automated.

Law that creates obstacles to automating operation and generation of software—any software—impedes the key means by which computing progresses. If the class of software that is broad—as "intrusion software" as currently defined is, being essentially unapproved composition—then restricting automation of operation and generation of this kind of software is going to catch all the practical ways to make engineering progress in this software.

Essentially, such restrictions seeks to freeze the evolution and understanding of the so-called "intrusion software" in its present state. This will create gaps in understanding and ability between actors who can afford the chill and those who cannot, such as private parties, small companies, startups, and small groups of research, and individual researchers.

The proposed approach will fail to protect both security researchers and the basic conditions for progress in security engineering.

5 Chilling of innovation, a long-term take

In a long-term perspective, all innovative software is "intrusion software", inasmuch as it relies on composition. Composition is what people do with software from its inception to application; it defines all interesting systems. "Intrusion" is unforeseen, unexpected, or unapproved composition—otherwise known as innovation.

In the classic realms of expressive works—copyrighted texts, music, and other arts— "Fair use" is unapproved compositional intrusion on pre-existing material, and one of the fundamental exceptions to requiring prior approval. The realm of systems engineering needs a protection equally strong to evolve.

Engineers and researchers being liable for creating "intrusive" tools branded as violating copyright is seen as a chilling effect on innovation. Similarly, engineers and researchers working on techniques painted as intrusive should enjoy similar protections, for similar reasons. Construction of "intrusive" unapproved mash-ups should be no crime, but an ordinary and protected means of gainful employment (being, as it were, an engineering discipline right on the innovation trajectory).

The nature of engineering is creative reuse and pushing the limits, unexpected applications of existing products (not just ideas). Unapproved composition is at the heart of innovation.

Innovation is unapproved composition. In software, we know it as "exploitation". In software, any composition for which dedicated interfaces were not foreseen, pre-designed, envisioned, or provided is "exploitation". It's impossible for a designer to foresee all uses of a technology, or most productive uses, or even the primary use a decade from now – who could have predicted the WWW when designing multiuser machines? Inventors of the telephone envisioned its profit model as receiving information services from a central office, not as overwhelmingly a means of private conversations. When a monopoly manages to enforce an envisioned set of uses for an extended period, stagnation results.

In the case of security and privacy, stagnation at the current point would mean the status quo of ubiquitous insecurity and institutionalized imbalance of power between the state and the citizens, between well-funded

attack and resource-constrained defense.

Even though a Hollywood view of "exploitation" is that of enabling cinematic attacks, exploitation enables defense by orders of magnitude stronger.