Why Wassenaar Arrangement's Definitions of Intrusion Software and Controlled Items Put Security Research and Defense At Risk—And How To Fix It

Sergey Bratus, D J Capelis, Michael Locasto, Anna Shubina

October 9, 2014

Abstract

In this article we argue that Wassenaar Arrangement, as currently formulated, will have extensive harmful effects on computer security research and defensive software. We propose an alternative formulation that will achieve Wassenaar Arrangement's goal of protecting activists and dissidents in oppressive regimes without causing these chilling effects.

1 The intent of the Wassenaar Arrangement

The Wassenaar Arrangement's *intrusion software* clauses are intended to protect the activists and dissidents whose lives are endangered by government surveillance. The body of evidence that links persecution and computer surveillance is growing. The usual pattern of computing technology commoditization implies that this surveillance will grow in footprint and capacity while costs fall. The regulations of the Wassenaar Arrangement are intended to reverse or abate this trend, limiting the availability of computer surveillance to repressive regimes.

Unfortunately, as we demonstrate in this article, the Wassenaar definitions of *intrusion software* are overbroad, applying almost universally to elementary building blocks of security research. Among the unintended effects of the Arrangement's definitions are chilling effects on the development of anti-surveillance measures and on the discovery of existing vulnerabilities—and thus on fixing vulnerable systems. The Arrangement's definitions will impose a prior restraint on the publication of security research, analogous to the export controls on strong encryption software that were in effect in the 1990s.

The language of the Arrangement's definitions attempts to avoid these unintended effects by using explicit exemptions as well as a two-tiered structure of controls. This article demonstrates that these methods fail to cover the majority of technological artifacts and processes that are crucial to security research and defense, and are therefore insufficient to meet the intent of the Arrangement.

The anti-surveillance intent of Wassenaar will, however, be fully fulfilled if surveillance-enabling software and hardware were to be addressed directly. We propose such a direct approach: targeting *exfiltration*, which is a key part of surveillance, rather than the vague and overbroad *intrusion*.

In addition to the advantage of simplicity, this approach eliminates the potential ambiguity between the singled-out but not directly controlled class of *intrusion software* and its related classes of *controlled items* in the current Wassenaar language.

This document has the following structure:

1. The conceptual structure of the chilling elements in the current Wassenaar language is discussed in section 2.
2. The overbreadth of these elements is discussed in section 3 and appendices A, B, and C.
3. Section 4 proposes replacing the key concept of *intrusion software* with *exfiltration software*. This proposed replacement addresses the Arrangement's stated intent and avoids the unintended chilling effects.
4. The article concludes with a forward perspective on the regulation of independent security research, and an argument that such regulation must exercise caution in order to preserve the *citizens' science* nature of such activity.

2 Definitions of intrusion software and controlled items in Wassenaar Arrangement

The Wassenaar Arrangement (WA) uses a two-step conceptual structure to define the surveillance-related software it purports to control. First, the WA introduces the concept of *intrusion software*, defined 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.

The class of software defined by the previous quote is both broad and fundamental. As demonstrated later in this article, the class covers not only software known in computer security jargon as *exploits* and *rootkits*, but *all elementary means of software instrumentation, construction, and deconstruction* beyond the interfaces provided by the software's pre-defined interfaces—despite explicitly excepting "hypervisors, debuggers or Software Reverse Engineering (SRE) tools" in the note to the above definition.

After defining intrusion software, the WA does not directly control this new class. Instead, the WA 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.[1]

The following elements are subjected to particular control:

> 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 *per se* would inhibit communication between malware researchers and discovery of new vulnerabilities, a concern the authors of this article agree with. Controlling the second class, derived from the first, purports to limit the scope of the WA controls to the means of developing and delivering malware.[2] Unfortunately, this definition is still overbroad and will chill both basic and applied security research, as we explain below.

3 The problems with the Wassenaar Arrangement approach

Unfortunately, this two-class structure creates more problems than it solves. The so-called intrusion software class covers common and essential software techniques used throughout software engineering, not just potentially nefarious ones unique to malware and attack tools. In fact, these techniques are used by computer security products, remote management software, antivirus, enterprise reliability and monitoring, and operating systems.

Although this class of software is not directly controlled by WA, the software used to develop, generate, automate, and deploy it *is* controlled. This creates a huge potential for unintended consequences, since automation of development, analysis, and deployment is the primary way of making progress in software engineering, including but not limited to improving software reliability and security. Any non-nefarious software kinds and techniques deemed intrusion software under WA will have tools to improve their reliability and security controlled—and chilled.

[1]Further details can be found in `http://dymaxion.org/essays/wa-items.html` by Eleanor Saitta. We would like to thank the author for helping us understand the WA's structure of controls.

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

The WA-defined intrusion software class is extremely broad. Centerpiece of the WA definition of intrusion software is "modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions." WA construes this behavior as a sign of nefarious intent, intrusion. However, such modification, is, in fact, *common and essential* for many software engineering techniques. Far from being unique to malware or to attack tools, the same techniques are used by remote management software, antivirus, enterprise reliability and monitoring, and operating systems. There techniques are to software engineering what saws, hammers, and planes are to carpentry; they are ubiquitous and indispensable.

Simply put, WA definition makes the mistake of equating techniques embodied in software with just *one* of its potential uses. This leads to a quagmire of unintended consequences.

Exceptions written into WA for this class do not help, because they exempt only a few kinds of tools among many kinds that are critical to security research and development. WA's definition of intrusion software covers not only potentially nefarious software but also *all elementary means and techniques of software instrumentation, construction, and deconstruction* outside of the software's own pre-defined interfaces. Since modern security software embodies complex combination of many techniques, a short list of exempted products will not be adequate to stop unintended consequences.

One strong unintended consequence will be complicating the development of defensive software techniques and products.

For example, the purpose of the popular *Detours* software library from Microsoft is to intercept and modify standard execution paths of software. Thus Detours squarely fits the WA's definitions of *both* intrusion software and controlled items for developing such software. Yet Detours is a key industry tool for dynamic security patching of software, for monitoring software, and for debugging it. Many similar libraries exist for other operating systems and platforms, and others are currently being developed for new platforms such as smart phones.

As another example, the first personal firewall for Windows, the pioneering security product *Black-Ice*, installed itself by modifying the standard execution path of the Windows operating system. This technique made its installation easy, and contributed to the success of the product; it also allowed many users to secure their machines with the 3rd-party software when no solution from Microsoft was available. Outpacing vendor support is common for 3rd party software solutions.

In Appendix A we explain these technical cases in detail.

Controlling technologies of software development and automation is extremely broad and contrary to principles of software engineering. By starting with an overbroad definition of the *intrusion software*, the Wassenaar Arrangement subjects an even broader class of software tools to unwarranted control. Namely, WA's control lists, as per items 4.A.5, 4.D.4, and 4.E.1, cover the automation of *development*, *generation*, and *operation* of software elements defined as intrusion software.

Yet automation is the primary means of software engineering and research. The absolute majority of modern software is *generated* with automation tools like compilers and linkers, which use development technologies such as templating and scripting. Modern software is also *operated* by means of automation tools. For example, the Detours library we mentioned above fits both definitions.

Moreover, the most promising modern Computer Science approach to analyzing vulnerabilities in software and to prioritizing them to be fixed by their potential input relies on techniques for Automatic Exploit Generation (AEG), which we cover in Appendix B. Yet all software tools and techniques involved would fall under the WA controls.

Keeping actual malware samples or exploit payloads off the controlled lists may seem like a way to allow security researchers and defensive software developers to communicate unimpeded. However, nowadays effective communication requires exchanging recipes, tools, and frameworks that *automate* finding of vulnerabilities and *generation* of exploits—exactly the items controlled by WA. Thus WA fails in its apparent intent to prevent chilling security research. We discuss this in detail in Appendix C.

Once we recognize that the class of software that forms the base of WA definitions is overbroad and must not be chilled for the sake of research and deepening our understanding of software flaws, then we must also recognize that the very means by which this understanding is advanced—automation of generation and operation—must likewise not be chilled.

We note that slowing or halting the progress of software engineering favors existing powers in the field, protecting them from the disruption of smaller private parties. This ultimately defeats the 3rd-party providers and the citizen's science of security solutions to protect the dissidents and activists that WA seeks to protect.

4 Fixing Wassenaar: How to control surveillance without chilling security research

In this section, we outline our proposed approach to fixing the Wassenaar language, fully implementing its intent, and avoiding the chilling of computer security research.

Surveillance is exfiltration. Simply put, intrusive surveillance targeted by WA requires the surveilling party to receive sensitive data from the compromised computer. Without such receipt, surveillance cannot be said to have occurred. In nearly all of surveillance scenarios, surveillance software sends sensitive data to a command-and-control center operated by the government. The sending and the collection of sensitive data occur without the users' knowledge or permission. In other words, surveillance software critically depends on its ability to secretly *exfiltrate* data from the computer, without user permission or knowledge. Without this ability, surveillance effectively cannot exist. [3]

Exfiltration is thus key to surveillance. Controlling exfiltration functionality will therefore effectively control surveillance software. Hence we propose replacing the overbroad concept of *intrusion software* with *exfiltration software*, which we define as follows.

> *Exfiltration software is:*
>
> *1) Software designed to transmit data it did not create, or derived from data it did not create, except when any of the following conditions are met:*
>
> *a) The creator of the data provides informed permission to the software to transmit the data.*
>
> *b) A user or administrator of the computing system provides informed permission to the software to transmit the data.*
>
> *c) Systems software set up by a user or administrator of the computing system provides the data to the software under the design of the computing system as part of routine and expected behavior.*
>
> *2) Software designed to transmit data from the network in which it is contained to an external network, when installed or operated without direction of a user or administrator of that network.*
>
> *In the above definition, informed permission is permission explicitly given by a user or administrator through an interface that clearly communicates the intention and scope of the access, as well as all recipients of the transmitted data.*
>
> *In the above definition, data includes but is not limited to messages, images, files, video and audio recordings, as well as data streams from the computer peripherals such as camera, microphone, and various sensors such as GPS, accelerometer, etc.*

Since programs continually transmit data and create data, care must be taken that this definition is not overbroad like the definition of the *intrusion software*. In particular, this definition should not interfere with or burden legitimate advertising activities, legitimate software or hardware fingerprinting activities, and other legitimate data gathering activities. In the following sections, we show that these legitimate applications are not in danger.

4.1 E-commerce sites not in danger

E-commerce has long relied on *cookies* as a mechanism for maintaining web sessions. *Cookies* were introduced as early as 1994 in the Mosaic Netscape browser as a mechanism to support the concept of a session, that is to say, to allow the server-operating vendors to connect together all requests from the same web user, and to create the user experience of a continuous history of interactions, even though each web request was completed on its own and separately from others.[4] Flash and HTML5 introduced

[3]The only remaining surveillance option would be to secretly accumulate surveillance data on the device itself, which is impractical for large amounts of data, prone to detection by the user, and requires physical retrieval of the user's device. Most importantly, this option fails to select targets out of the general population, which is the predominant use of surveillance by repressive regimes, and will remain so.

[4]From `http://en.wikipedia.org/wiki/HTTP_cookie`:

> The term *cookie* was derived from *magic cookie*, which is the packet of data a program receives and sends again unchanged. Magic cookies were already used in computing when computer programmer Lou Montulli had the idea of using them in web communications in June 1994.[8] At the time, he was an employee of Netscape Communications, which was developing an e-commerce application for MCI. Vint Cerf and John Klensin represented MCI in technical

analogous mechanisms. These *cookie* and *local storage* data are created by the e-commerce vendor server software from the data transmitted to them by the client browser software.

The intent of these mechanisms is to maintain the identity of an e-commerce customer across a series of transactions between the customer and the e-commerce vendor, pursuant to the customer's intent of completing the e-commerce transaction, and to the customer's intent of maintaining a record of such transactions. Notably, the customer's identity tied to the payment method is explicitly provided to the vendor at customer sign-in time, since payment and disbursement of goods is only possible via a confirmation of customer identity.

Importantly, the decisions on what data to transmit and what data to store reside exclusively with the client-side software. Thus vendors of client software such as browsers bear the onus of protecting the users from inadvertently disclosing their protected private data. Indeed, these vendors responded to the user demands for protecting such data with introduction of such features as *private browsing* that explicitly disassociates users from the stored records of their previous transactions.

In summary, e-commerce software that acts within the scope of information explicitly provided by the users will not be chilled by the above controls.

4.2 Web ads not in danger

Web advertisement industry has long relied on collecting information pertaining to user visits to commercially operated websites, even though those visits have not resulted in an actual purchase—such as user searches for particular merchandise.

However, the data transmitted by the browser programs in the course of these visits is derived solely from the explicit user inputs, and from the program's inherent properties such as its version and supported communication formats, which are not protected private data according to the above definitions.

Thus targeted advertising based on user inputs is not chilled by the above controls.

4.3 Android, iPhone apps not in danger

Many applications of modern smart phones such as Android or iPhone depend on accessing potentially sensitive data such as the phone's location. However, both iPhone and Android apps request the user to explicitly approve their permissions to access such sensitive data throughout the app's lifetime. The user must grant these explicitly enumerated permissions before an app can install.

Thus development and profit models of Android and iPhone apps are explicitly exempt from the above controls.

4.4 Advanced browser fingerprinting not in danger

Recent research demonstrated that different versions of browsers implementing the HTML5 specification can be distinguished by how they process certain crafted drawing requests from the server. [5] In particular, researchers from University of California, San Diego discovered that the differences in rendering of certain curves and patterns defined by the same mathematical formulas are enough for a server to distinguish between browser versions. [6] Such research is important for preserving user privacy in the world of ever-increasing software complexity, since it anticipates how such complexity can be used or abused on the Internet.

However, all such observable differences stem entirely from computations performed by the browser programs themselves, without accessing any of the user-entered data whatsoever. Thus research into enumerating these differences will not be chilled.

discussions with Netscape Communications. Not wanting the MCI servers to have to retain partial transaction states led to MCI's request to Netscape to find a way to store that state in each user's computer. Cookies provided a solution to the problem of reliably implementing a virtual shopping cart.[9][10] Together with John Giannandrea, Montulli wrote the initial Netscape cookie specification the same year.

[5] "The Web never forgets: Persistent tracking mechanisms in the wild", Acar et. al. `https://securehomes.esat.kuleuven.be/~gacar/persistent/`

[6] "Pixel Perfect: Fingerprinting Canvas in HTML5", by Keaton Mowery and Hovav Shacham, `http://cseweb.ucsd.edu/~hovav/papers/ms12.html`

5 Protecting whistle-blowers

Whistle-blowers inform the public of abuses they encounter in the course of their employment by or business relation to the abusing organizations. To succeed, these whistle-blowers must provide credible evidence of abuses. Since the workflow of most organizations has by now been heavily computerized, there is a legitimate concern that abusers will use technological means as obstacles to whistle-blowing. Although so far the balance of power has favored the whistle-blowers, concerns remain that future technological developments will tip this balance. Consequently, software that might help whistle-blowers to expose evidence of abuses that they become privy to in the course of their business relationships should not be chilled.

We note that the proposed language preempts these concerns, since any data access in the course of a computerized business process is by definition approved by both the user and the owner of the involved computer device. While laws and contracts outside of the technology realms might govern disclosure of the data that is accessible to employees in the course of their employment, explicit access permissions must be set by administrators on behalf of employers, and are exercised and affirmed every time data is accessed.

Thus whistle-blowers are explicitly protected by the proposed definition.

6 Looking back to the 1990s "Crypto Wars"

The 1990s were a formative decade for the commercial Internet in the US. Unfortunately, during this same time the US government policy was to treat strong encryption as a threat and to control implementations of certain cryptographic algorithms as munitions, subject to vigorous enforcement of export regulations. In 1993 the author of the original PGP software, Phil Zimmerman became the target of an FBI investigation for munitions export without a license. This investigation lasted till 1996. At the same time a series of failed technological "solutions" and mandates, such as the backdoored-by-design Clipper chip[7] and third-party key escrow were promoted as a legally safe way for telecommunications industry to implement compliant encryption—which would have essentially amounted to pretend security.

Export restrictions on artifacts of cryptography have doubtlessly harmed its practical progress. Not only Johnny Q. Public still can't encrypt[8], but John the Special Agent can't encrypt either![9] No matter where one stands on whether and how much the latter should be allowed to wiretap the former, John certainly has things to hide and in fact a duty to hide them—in which he is conspicuously failing.

Could it be that *both* of these failures are due to the fact that deployment of strong cryptography was stymied just when today's dominant communication protocols and infrastructure were rapidly developing? The fact is, they ended up leaving cryptography behind, and matured without incorporating cryptography at their core. Superiors of John the Special Agent may have had visions of him using separate, special technologies vastly stronger than Johnny Q. Public's and obtained from sources untainted by the weaknesses of public commodity communications; it appears this vision was wishful thinking.

If having to pretend that poor cryptography was secure because practically exploring stronger cryptography was a legal minefield led us to this point, where would pretending that computers are secure because of a likely minefield arising in exploitation engineering lead us from here? It will likely be worse, because the field of cryptography by 1990s already had mature mathematical theory not easily undercut by the drag created on its engineering practice. Systems security, on the other hand, is only building up its theoretical foundations, and is in need of much more feedback and generalization of its practice.

If the practice of exploring the programming of programs' faults becomes subject to regulation as vigorous as the so-called 1990s *Crypto Wars*, will this practice develop enough to warn us before unsecurable designs come to dominate in critical infrastructure, power management, medicine, or even household appliances beyond any hope of replacement? Will we be surrounded by an Internet of Untrustworthy Things just as we are surrounded today by an Internet of Things that Can't Keep a Secret (or at least are no help to an ordinary person for doing so)?

[7]M. Blaze. "Protocol Failure in the Escrowed Encryption Standard." Proceedings of Second ACM Conference on Computer and Communications Security, Fairfax, VA, November 1994

[8]`www.usenix.org/events/sec99/full_papers/whitten/whitten.pdf`

[9]`http://www.usenix.org/event/sec11/tech/full_papers/Clark.pdf`

7 Chilling of innovation, a long-term take

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

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.

A Why the WA intrusion software definition has wrong granularity that exceptions cannot fix

The WA defines *intrusion software*, and thus by derivation 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, the operations in the definition are 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 7). These operations are critical to the performance of state-of-the-art security research, as well as to other kinds of technological progress in software engineering. These operations are especially critical for improving defense. At the same time, the exceptions to these definitions (”Hypervisors, debuggers or Software Reverse Engineering (SRE) tools; ...”) are at a different, much higher level than *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. The WA definitions whitelist a particular set of combinations and compositions that are seen as important to software engineering *today*, but does not and cannot include the set of all such combinations and compositions that will become equally or more important in the future.

The excepted programs or products contain both components with functionality labeled *intrusive* as well other kinds of 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, in observability, and in security of software, is made. For example, Microsoft’s release of its Detours library was a significant step forward. As described by Microsoft, Detours is both *intrusion software* and a *controlled item* by the language of the Wassenaar Arrangement.

> 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. Hooking is used for debugging, instrumentation, and performance tuning of software. Hooking is also used for patching vulnerabilities in software, as well as to upgrade software that lacks a dedicated upgrade mechanism. This latter function is a critical security function for legacy software, including the 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 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 fit the WA definition of controlled items, since they operate, manage, and automate the application of the means by which Detours modifies the execution path.

A.1 "Standard execution path" contradicts the nature of modern software design

WA language depends on the concept of the *standard execution path* of a piece of software. The implication is that for every piece of software or at least for most important kinds, a set execution path exists, and modifications of it are suspect and likely onerous.

However, modern software engineering in fact emphasizes customization of software, which explicitly modifies the standard, as-shipped execution paths! For example, the popular Firefox web browser includes a mechanism for modifying most aspects of its functionality (that is, its execution path) with the so-called Add-ons, which interleave *externally provided instructions* with the main Firefox code logic. Over ten thousands of Firefox add-ons are available at the time of this writing from the official Firefox vendor site alone. Similarly, Google's featured browser, Chrome, includes an analogous mechanism.

Software customization mechanisms for modifying the *standard execution path* of the software that forms the basis of the Internet as we know it are not limited to web browsers. For example, the Apache web server that, according to Netcraft web surveys, is the leading software behind roughly 40% of all Internet web sites, ships with a similar customization mechanism.

In fact, the ability to modify the standard execution path of programs is the basis of an entire new programming paradigm, Aspect-Oriented Programming. Popular programming languages such as Ruby and its Ruby-on-Rails leading web development environment, implement this paradigm in the form of *mix-ins*, a standard language mechanism.

Thus the WA emphasis on *standard execution path* is misleading, and clashes with the trends of modern software engineering. One way to reconcile this language with these trends is to assume that *standard* in fact has the much narrower meaning of *expected by developers*. We make this assumption in the following analysis.

A.2 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.[10] 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.

[10] `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 chilled by WA language.

"Defeating protective countermeasures" is not a meaningful way to characterize software. Protective countermeasures are no different from any other obstacles to exploitation; it does not matter in the final security analysis whether, e.g., the attacker cannot control the computation flow because the memory corruption afforded by a bug is serendipitously not extensive enough or because a protective measure somehow captures the attempt. In either case, security analysis deals with a system of constraints; it doesn't matter where constraints come from, or even whether they are external or internal to the target system. Separating these constraints by origins would merely confuse and weaken development of rigorous analysis.

Moreover, the intent to bypass countermeasures is neither obvious nor easy to argue. A piece of malware—or a defensive 3rd party security product like the pioneering BlackIce product described below—may use a specific method of altering the target software either because that method is more reliable or because the original vendor blocks some simpler methods. Intruding equals composing in every technical sense.

A very good discussion of this topic can be found in Rob Graham's story of how he built the first personal firewall BlackIce: `http://blog.erratasec.com/2013/03/the-debate-over-evil-code.html` Rob was able to inject his protective code (in WA terms, "intrude") on the Windows operating system manually, but today a similar composition effort to harden, say, an iPhone, would notably require automation, which would fall into controlled lists of WA.

B 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.

C 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 vulnerability research in particular.
2. Its practical application to actual research and engineering artifacts used in vulnerability 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* (VM) programs, such that run automatically generated hybrid *bytecode* commands for Java and .Net programs and at the same time *operate* them.[11]

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 incorporating the functionality to automatically generate what used to be manually written code in the main program's 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.

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

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 vulnerability 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 vulnerability 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 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.

Fuzzers: a highly practical approach to discovering vulnerabilities, threatened. A necessary requirement for software to be considered trustworthy is that the software operates predictably and as expected no matter what inputs it receives. This requirement is especially important when the software receives inputs that can be maliciously crafted by attackers (which is, e.g., the case for any software that receives its inputs from the Internet). Unfortunately, software engineering and development practices are not yet at the point when this requirement can be assured.

As a result, an effective method of discovering security vulnerabilities in current software is to supply that software with a series of crafted, invalid inputs and to observe which inputs cause unexpected effects such as crashes or modifications of the typical execution paths. Such inputs are referred to as *fuzzed* inputs by security practitioners, and the process of generating these inputs and observing their effects is referred to as *fuzzing*. The software that automates this process is called a *fuzzer* or a *fuzzing framework*.

Fuzzing is by now the subject of several industry textbooks,[12] a large number of research papers, and an integral part of the secure software development lifecycle. Major vendors of software and hardware such as Microsoft and Cisco use fuzzing in their software development and testing processes.

However, fuzzers and fuzzing frameworks fall within the WA definition of the controlled items, because they automate a crucial step in development of *intrusion software*: finding the potential points where the *standard execution path* of the target software can be modified, and logic external to the target program purpose can be introduced.

Until new emerging methodologies for engineering input-handling code become a universally accepted industry standard, fuzzing will remain a key technique for software security testing. Controlling the development of new fuzzing techniques and of software that implements them will set back the existing industry practices by years if not decades.

Vulnerability-finding tools must generate "intrusions" to be effective. To fulfill their business purpose, vulnerability-finding tools such as fuzzers must generate *complete* recipes and payloads for "modification of the standard execution path" that would put a target program or process under attacker's control. Stopping short of producing and testing such recipes would hinder a necessary business function of the tool: prioritizing vulnerabilities into those actually exploitable under constraints imposed by defensive measured such as DEP and ASLR, and those not exploitable. To effectively allocate the inherently limited specialist labor to fixing the actually exploitable vulnerabilities first, a vendor must receive evidence of exploitability. Since bugs in complex software systems are plentiful, following this evidence-based approach is a virtual necessity for software vendors.

Moreover, the ability of fuzzers and other automated security tools to find vulnerabilities depends on their ability to exercise all code paths possible in the target software, *including those not normally exercised*. In other words, this ability depends on finding precisely those execution paths that an *intrusion*

[12]E.g., "Fuzzing: Brute Force Vulnerability Discovery", by Michael Sutton, Adam Greene, Pedram Amini, Addison-Wesley Professional, 2007; "Fuzzing for Software Security Testing and Quality Assurance", Ari Takanen, Jared DeMott, Charlie Miller, Artech House, 2008.

software would use. A fuzzer is essentially a tool for automatically generating recipes for triggering these paths.

Consequently, vulnerability analysis tools developed towards algorithmically automating security analysis and striving to automate judging of a bug's exploitability. Modern research tools such as EXE[13] and KLEE[14] use sophisticated static and dynamic automated analysis methods; similar proprietary techniques are used in industry by Microsoft and other major vendors.

Can a fuzzer avoid being controlled by stopping short of producing a complete intrusion or exploit? Only at the cost of ignoring state-of-the-art research. Early fuzzers indeed stopped at producing recipes for triggering bugs that led to crashes, and left the rest of the exploitability analysis to costly manual analysis by experts. As a result, these early fuzzers tended to produce more leads than defenders could investigate, nor provided defenders with any actionable prioritization between the triggered bugs. Such levels of performance is currently neither acceptable nor scalable, with few exceptions.

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.

In summary, the current Wassenaar approach will fail to protect both security researchers and the basic conditions for progress in security engineering.

[13] "EXE: automatically generating inputs of death", Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler, 2006, In Proceedings of the 13th ACM conference on Computer and communications security (CCS '06). ACM, New York, NY, USA

[14] "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs", Cristian Cadar, Daniel Dunbar, and Dawson Engler, 2008, In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, Berkeley, CA, USA