Avoiding a War on Unauthorized Computation: Why Exploit Regulation is the Biggest Danger to Coder Freedom and Future Security

Sergey Bratus, Anna Shubina

DRAFT

In recent cyber-security discussions, activists compared exploits with weapons and called for legal intervention and regulation in the name of protecting user security and privacy. We argue that any attempts to regulate – or, indeed, legally define – exploits will in fact cause irreparable harm to both coder freedoms and actual trustworthiness of consumer systems. The only technical difference between exploit programming and "normal" programming is that the former creates computation that is not anticipated by the target's engineers, undesirable to vendors, and depends for its reliability on a deeper understanding of the target's actual rather than documented engineering composition. Restrictions on activities of exploit programmers will inevitably burden all other kinds of unexpected or unapproved programming-related activities, such as jailbreaking. We argue that "war on unauthorized computation" will not improve trustworthiness of our systems; to the contrary, it will reduce the sum of our knowledge about how they can and cannot behave – and thus of what they can and cannot be trusted with.

1 The challenge of exploits

The current discussion about regulating exploits so far revolved around scenarios of rogue governments acquiring instant, push-button power for taking over systems. This scary scenario creates an instant desire to prevent it, and limiting creation or transfer of these dangerous artifacts naturally springs to mind, hence calls for regulation of "weaponized exploits". Throughout these discussions, "exploits" are conceived of as easily identifiable artifacts of a specialized craft rather than a general computing phenomenon.

However, whenever there is a call to regulate a programming activity – and exploits are clearly programs, whose developers use sophisticated software engineering tools – we should be aware of unexpected consequences of such regulation. For example, constraining the ability of developers to be paid for their work will certainly curtail the time they can afford to spend on it, thus restrictions on exploit purchases will translate to restrictions on exploit research

and development as well. We should make sure that we clearly define that which would be restricted – and consider both the direct scope of such regulation and the inevitable elaboration and expansion of its underlying legal definitions and theories.

We want no restrictions on security research that exposes vulnerabilities in existing trusted systems – if anything, we want *more* of it to happen, in particular to offset the above rogue government abuse scenarios. Furthermore, we do not want to accidentally restrict the freedom of general programming activities, even where they might be construed to contribute or be related to exploitation – e.g., when they require reliable "jail breaking" or rely on reverse engineering that needs to actively modify its running target.

Thus, should we want to prevent the "cyber-weapon"-based scenarios with regulation or law, we must consider the challenge of defining what constitutes the exact kind of these "weapons" that we want to impede, and exactly how they are different from other phenomena of applied computer security research, general programming, even daily computer use. Without such clear boundaries and definitions, resulting laws and regulations are likely not only to be ineffective but enormously counterproductive to us being able to eventually trust the integrity of our computers. Ultimately, we may find – and, we argue, we will find – that such boundaries on computing artifacts are impossible to draw without impeding a whole spectrum of activities critical for both programmer freedoms and advancement of security.

The rapid expansion of activities covered by the revisions of Computer Fraud and Abuse Act (CFAA) provides a cautionary example. Having started with a narrow scope of protecting US government computers, the law grew to cover nearly every access to every computer around the world, at the same time reaching unconstitutional vagueness in what exactly it criminalizes [4]. There is every reason to expect a similar expansion in regulation of exploitation-related activities.

Another important example of a law that had significant chilling effects on security research is the Digital Millenium Copyright Act (DMCA). While it apparently fell short of its intended goals (as evidenced by much broader laws such as SOPA and PIPA subsequently lobbied for by copyright owner associations), the DMCA resulted in legal threats against security researchers engaging in reverse engineering that included "technological protections of content" – no matter how inadequte these protections (e.g., arrest and prosecution of the Russian researcher Dimitry Sklyarov for exposing multiple weaknesses of Adobe DRM schemes) or how critically important the analyzed systems were (e.g., legal threats to silence public debate about the trustworthiness of electronic voting machines[1]). The continuing chilling effects of the DMCA are covered by many resources on the web, in particular EFF's *www.chillingeffects.org* and the *Freedom to Tinker* blog, which frequently features contributions by Prof. Edward Felten, who had to fight DMCA-based legal threats to his security research in court in the landmark *Felten, et al., v. RIAA, et al.*

[1] `https://www.eff.org/cases/online-policy-group-v-diebold`

2 What is an exploit, or "I'll know it when I see it"

Exploits made an impressive appearance in Hollywood hacker movies: a hacker runs a program and is immediately granted access to vital facilities. Exploits have made attending-grabbing headlines, which propelled them into the current debate. There is thus a danger that instead of a general and broadly important computing phenomenon the debate will focus on the media artifact – to the detriment of computing.

2.1 Flawed representation of exploits

In the ongoing discussion about regulating exploits, exploits to be regulated are viewed as unfailingly effective products that give their wielder instant power to take over systems silently and reliably – a scenario that tends to occur in hacker movies rather than in reality.

When exploits are discussed in this manner, we end up with the image of an exploit as an artifact rather than a general computing phenomenon. In particular, in the "surveillance argument", which we discuss below, exploits are clearly conceived of as easily identifiable artifacts of a specialized (and ethically questionable) craft rather than expressions of research of fundamental importance to security of computers in general. This is a dangerous misconception.

2.2 Technical reality of exploits: defining "evil computation"

Despite being one of the most frequently used words in the security industry vocabulary, *exploit* has no formal definition, nor did it need one while it was being used by people who shared the same intuitions about security and exploitation. Hardly anyone remembers who coined it; "it's always been there".

This lack of common definition presented a subtle trap when the word "exploit" started to get used by journalists and activists - lacking a definition to correct themselves, they started using it in the sense "something that does bad things to a computer".

To define an exploit, we need to start with the basics.

Since exploits exist for all levels of software targets from chip firmware to office document applications and web-based e-mail and content management systems, any legally actionable definition of an exploit will inevitably grow to be general enough to cover all major categories of computing systems. The common feature that unites all levels of exploitation is that *the exploit makes the actual computation happening on the target deviate from the one expected or designed, in a way that is reliably controlled by the exploit.* It is highly likely that the working legal definition will eventually capture this common feature, although one can only guess at the possible legal formulations, with the evolution of the CFAA of serving as a possible pattern (one could also recall the evolution

of "actionable copy", culminating in the theory that "a copy in RAM" was one under the DMCA).

It should be noted that an exploit may or may not be automated; important parts or the entirety of it may be a recipe for user action. Moreover, exploit computations may require additional physical conditions to succeed, e.g., wrong voltage, exposure to heat, or light. An exploit may also depend on a heavy or varying CPU load, or the ability to create such load on the targeted system, e.g., to create a side channel or a race condition. Such manipulations, dating back to MLS covert channel concerns, are gaining renewed relevance in cloud environments. For lack of space we do not further discuss such scenarios, but note that they would easily fall under any broad enough prohibitions of unexpected computation that would target software-only exploits.

In the common experience of software and hardware engineers, deviations from expected execution quickly bring the system into a corrupted, inconsistent state, in which it violates one of the hardware-enforced limits and either crashes or is terminated by some monitoring component. An exploit, however, takes care to minimize corruption and restore consistency, and controls the execution so that no fatal violations occur, as if the target was operating normally. In short, the exploit acts as a new program for the target – one that its original engineers would consider impossible or really strange, but an effective and reliably executing one nonetheless.

The fact that **an exploit acts as a program for the target** is extremely significant, and here is why. Attacking exploits as a general phenomenon can only be done by attacking *unexpected or unintended computation* brought about by *programming a computer by non-standard or unanticipated means*, because such computation is what all these different exploitation scenarios have in common. However, taking the view that it is illegal to either divert control within a program from its intended paths or prevent it from crashing after an error condition in it has been triggered is clearly over-reaching, because bona-fide runtime patching, error recovery, and debugging depend on the exact same techniques. Outlawing particular ways of triggering program errors would be even worse, because testing depends on them. In short, attempting to outlaw exploits as a computing phenomenon will have broad repercussions to the fundamental practices on which the hope of trustworthy programming depends; ironically, it will lead to programs that are harder to debug, test, compose, and understand.

In the end, as we try to find a common definition to fit all exploits, we are left with *the escape from the platform's intended or expected limitations.*

2.3 Limitations and escapes, a developer's view

When securing a system, its developer, vendor, or operator may rely on existing or assumed properties of its underlying software or hardware blocks, such as the programming environment's language and runtime safety properties (e.g., type safety, memory management features, etc.), OS features (e.g., separation of userland and kernel contexts, process isolation), chipset protection bits and fuses. The developer or vendor may also deliberately limit access to the system's

functionality or programmability, e.g., by "jailing" third-party development or hiding programmable interfaces from it, allowing only signed code to load, and so on. A vendor's security model may be tied to its business model more or less closely and dictate the nature of limitations from which an exploit demonstrates both the possibility and mechanism of escaping.

From a developer's or vendor's point of view, exploitation is *any* undesirable execution in a software product that they did not expect to occur – whether it is related to undesirable modifications of the product, unlocking extra functionality, circumvention of a DRM scheme or some other intentional limitation, unintended instrumentation, reverse engineering – or is a larger security issue that should lead to rethinking the platform's design for users to be able to trust it. Arming software vendors or owners with legal tools and theories to suppress unintended programming of their products will likely have chilling effects beyond the infamous anti-circumvention provisions of the DMCA and is unlikely to improve actual security of products.

Whereas some escapes from the system's limitations merely expose an engineering oversight, fixable in the next product, others – incidentally, those that push the state of the art in exploitation and receive accolades at security conferences such as BlackHat and Defcon – tell us more about the platform than any vendor's documentation. These escapes typically expose false assumptions of what computation is and isn't possible to achieve on a platform, engineering misconceptions that may then take up to a decade to correct provided significant, dedicated vendor effort.

Convincing the vendor that any given problem is serious or possibly a matter of faulty design cannot be accomplished with a one-person one-off ("works on my computer") proof of concept, since any single bug can be papered over and declared fixed, merely breaking the proof of concept without understanding its larger implications. The exploitation technique needs to mature and become known and reproducible. A body of knowledge and labor are needed to expose the fact that the problem is general.

It is easy to blame developers for being slow to fix problems as they wait for more evidence of the problems' generality and not react to proofs of concept, but they could be said to merely follow the evidence-based approach, where reproducibility of results and accumulation of evidence take their time. For example, faced with mounting evidence, Microsoft in 2002 established its "Trustworthy Computing" initiative, hired prominent vulnerability researchers, and eventually incorporated defenses into products. Notably, many of these defenses (e.g., DEP and ASLR) were first prototyped in hacker research as far back as 1990s (in particular, by pioneering hacker projects such as PaX and OpenWall), and in fact co-evolved with exploitation techniques. Even more notably, the proper understanding of these techniques' scope and applicability by industry only emerged eventually, guided, yet again, by exploitation developments. For example, when first introduced in 2004 Windows XP Service Pack 2, DEP was described as protection against "buffer overflows", even though exploitation of this type of vulnerability that did not involve any executable payload has been known since 1997; ASLR features were later brought in to

partially address it. Now these lessons learned from DEP and ASLR inform the design of every modern OS and have become the de-facto industry standard, transcending a particular vendor or platform – thanks to continuing evolution and generalization of exploitation.

The above example shows the importance of free co-evolution of exploitation and defense, in which exploitation must be free to attract resources and talent. Limiting exploitation will not improve security, but rather help preserve the narrow, pigeonholed view of flaws, leaving them with fixes that do not fully address the underlying problems.

Neither do all problems that exploits expose represent mere engineering challenges. To see this, we next take a look at computer science theory behind the phenomenon of exploitation.

3 Exploitation and Theoretical Computer Science

While classic computer science concerned itself with answering the question of which problems could and could not be solved with a computer (including which behaviors could and could not be programmed), hacker research pursued the same questions with a different emphasis: an empirical study of what actual physical systems could and could not be made to do. Exploitation has been the tool of this research and produced results that are moving us forward to computers we may eventually be able to trust.[2] Interfering with this research now may rob us of a future theoretic generalization of these results.

State-of-the-art exploitation techniques of today will be key case studies for computer security formalisms years from now. Today's academic research efforts in abstracting exploitation techniques, automatic finding of vulnerabilities, and checking exploitability of bugs are typically informed not by the latest advances in exploit practice, but rather by exploit techniques that already reached maturity and have sufficiently clearly described in tutorials suitable for the general security community. The delay is not surprising and is partially due to the development of such tutorials being a non-trivial and time-consuming task, especially considering the difference between the classic curricula and exploiter experience.

One important example of this is "Return Oriented Programming" (ROP, see [6] for the historical account and references). Presented to academic audiences in 2007-2008, ROP changed the dominant academic view of exploitation from ad-hoc slipping of "malicious code" to the realization that exploitation involved complex programming models that manipulated only the target's own code and needed none introduced from the outside. This change was brought about by Shacham et al. demonstrating that exploit's Turing completeness in a typical target environment could be achieved without code injection. An im-

[2] I am indebted to Felix 'FX' Lindner for the description of the practical essence of InfoSec as "working towards computers we can finally trust".

portant innovation of this work was treating the phenomenon of exploitation as a computational one, and using computation theory terms to characterize its power. However, exploitation techniques involved were known and understood as suitable for generic target programming in the hacker community since at least 1997-2001, with detailed and general case studies published in 2000-2001. Thus accumulation of exploit case studies led to one of the most important paradigm changes in mainline academic computer security research: the change of the threat model from "malicious code" to "malicious computation" – a crucial (and much overdue) step in understanding the nature of insecurity and improving defense.

Of course, generalization and formalisms take time; moving beyond theory, constructing reliable access to the extra system/computation state is a hard engineering task, made harder by new system defense techniques. Restricting the sum of exploitation knowledge and effort now will hurt scientific insight years into the future.

4 Exploits: the engineering view

Exploitation depends on mapping out the actual limits on computation on a targeted system and fitting in unexpected computation within these limits. Clearly, exploit programmers must develop an intimate knowledge of the system's internals and architecture – and indeed, Phrack articles double as condensed introductions to most features of modern OS and programming languages.[3]

Thus exploit programming is a deeply structured engineering activity. In particular, each new exploitation technique includes at least two major engineering tasks:

(a) discovering and constructing a *programming model* based on the target system's mechanisms, features, and bugs, and
(b) constructing a reliable *composition* of the exploit program with its host system.

The first task is usually described as distilling the effects of a bug that can be reliably triggered with crafted inputs to a controllable minimum, a *primitive*. Together, primitives that read the exploit's crafted data play a role similar to that of a virtual machine's implementation of its pseudo-assembly bytecode instructions, while the crafted data act as a compiled bytecode program for that machine. Collective experience of finding and programming such environments within systems poses most interesting software engineering research questions – what design principles should guide future systems that wouldn't lend themselves to such manipulation?[4]

Exploitation is the evolutionary force that exposes and – when vendors and engineers are forced to pay attention – kills software designs too amenable to

[3] At Dartmouth, we've been using them as assigned reading in advanced courses; for details and papers see `http://hackercurriculum.org/`.

[4] We approach these questions in [9, 1].

exploitation. It is clear to many security practitioners that without continued co-evolution of exploitation techniques and defensive designs these questions will not be properly answered.

Once the promising bugs have been located, and a programming model is constructed, exploiter programmer's task shifts to locating all relevant system data objects and modifying them to allow the unexpected computation to run, while keeping other code from faulting or crashing on modified data. Essentially, these stages of the exploit bring in their own custom "linkers" and "loaders", which help *compose* the exploit's computation with the main system, or use the system's own symbol information and dynamic linking mechanisms – producing the finest descriptions of these mechanisms in the process.

The composition step cannot be skipped and cannot be replaced with some "thought experiment." Whether the needed compositional description of the target relative to the exploit execution model is obtained by static analysis of source code or binary, by dynamic analysis using such techniques as taint propagation, by reverse engineering, or by a combination of these, this is the most expensive part of exploit development. For example, in kernel exploitation [5, 7] the parts that deal with such composition dominate those that deal with the original vulnerability in both relative size and complexity.

Here we come to one of the most damaging effects of would-be exploit regulation on the future of security. In presence of modern OS defensive measures, the composition task presents considerable challenges, consumes the most effort, and provides the crucial evidence that the exploit is effective and does in fact contribute to the sum of knowledge about the target. This evidence is critical in convincing security-conscious vendor to take action, and, eventually, for having academic researchers to take note of the phenomenon – in the same way that a "thought experiment" differs from an actual physics experiment demonstrating a new phenomenon. Suppressing the funding of exploit development will destroy the distinguisher between the theoretically possible and feasible; the above analogy is even more relevant now that the cost of a physics experiment in qualified labor and tools is becoming comparable to that of an exploit against a modern system.

5 The surveillance argument

In some recent arguments, exploits have been described as a vehicle for government surveillance. However, surveillance software (or commercial spyware) should not to be confused with exploits. Such software might (or might not) use exploits for installing itself, but it may also be detected by integrity-watching methods informed by exploitation research. Installation of the very integrity protections that can detect surveillance agents *may require escaping from the vendor limitations on a device* – that is, exploitation.

Although exploit abuse by governments is possible, depicting it as a prime danger ignores both the nature of the exploit as a computing phenomenon and the fact that the bulk of Internet surveillance occurs without the use of exploits

– but with full architectural support from ISPs and networking vendors, due to the CALEA and subsequent legislation that mandated the embedding of "lawful interception" functionality into infrastructure-class equipment. Such mandated network-level support is by design invisible to the end-users, whereas exploitation of their diverse personal devices incurs considerable risks of failure or detection, as exploit programmers well know.

Moreover, even for on-device surveillance software law enforcement has a simpler path to surreptitious installation: subverting the platform vendor's primary trust mechanisms, such as code or site signing certificates (e.g., the FinFisher spyware). Dedicated, costly development of a remote exploitation "0 day" is pointless when physical access by government agents can be trivially obtained; mass deployment of "0 day" negates the advantage of it being actual "0 day".

Simply put, the connection between exploits and surveillance is weak; they are two orthogonal kinds of system manipulation. Ironically, development of methods to expose and analyze surveillance software (such as the German governments *Bundestrojan*, exposed in cases far beyond its original mission of terrorism-related investigation and analyzed by the Chaos Computer Club) may be curtailed by a "war on unauthorized computation". User modifications to commodity devices to make them less surveillance-prone, such as jailbreaking of smart phones (in which exploits played a prominent role) may suffer similar fate.

6 Conclusions

The unifying technical reality of exploits is *unexpected computation* and *unapproved composition* that a computing system can be made to carry out by feeding it crafted inputs. An actionable legal notion of a class of illegal computations and illegal compositions will inevitably lead to a totalitarian take on computing: the necessity to preemptively justify the legality of computation performed on a system, a presumption that anything that is not explicitly permitted likely falls into a prohibited category. For the sake of both keeping general "unapproved" programming activities free, and for the free scientific inquiry into actual software trustworthiness, we must avoid this trap – which may be the most subtle and, considering the good intentions currently driving it – the most dangerous yet.

Appendix A: Exploits from the point of view of Theory of Computation

Computer science studies the hard limits on what is computable (both in general and under specific limitations on resources or knowledge) by reducing the physical nature of the computing machines to simpler model machines that are more amenable to formal proofs. The tasks of interest are then translated as

inputs for these machines to consume; traditionally represented as a "tape" on which the inputs are encoded with symbols of some alphabet.

The most famous and powerful of these models is the Turing machine; there are many other more limited models such as finite state machines, pushdown automata of various kinds, and so on, progressively less capable in the classes of computational tasks they can perform. For tasks provably beyond a model's power, any solution claiming to be general will in fact fail on some input; for real systems this failure will translate into a bug, a failure to validate an expected condition – which may be exploitable.

Inputs consumed from the tape cause transitions in the internal state of the model, mirrored in its physical implementation as changes of variable values in a code module, or state of gates, or memory cells in a chip. Since exploits act as crafted input that drives the target system, the "impossible" computations they cause can and should be understood in the same terms (see [2, 3, 8]).

Existence of exploits indicates that the actual logical (in case of software) or physical (in case of hardware) target possesses *additional state and transitions* that are triggered by crafted exploit inputs. What engineers think of as their system is actually submerged in a larger system, with hidden state and transitions partially revealed by the exploit and lending the exploit its additional power to escape its limitations.

For software on a general-purpose computer this observation is, in a sense, trivial, since such computers have the power of Turing machines. However, this power is only gained by the exploit via a bug, a feature, or, more commonly these days, by a combination of features in the target. Famous exploits make a point of describing and then constructing a realistically *minimal* execution environment needed, which is non-trivial. Not surprisingly, finding whether a confirmed bug is exploitable is a hot research area.

Thus, from the theory point of view, exploits are programs for hidden machines immersed in real-world computing environments. Exploits expose differences between the target's presumed execution model and reality, in which computations deemed impossible by its security model actually exist. Describing and creating them is both programming and systems analysis; continued success of exploit programming schools suggests they follow a successful and teachable methodology.

Although this methodology and classic computer science concerned with limits of computation are currently far apart, this gap should be bridged if we are to eventually create more trustworthy computers. Needless to say, restrictions on what is essentially an empirical study of programming models' limits may be as devastating as restrictions on any other activity accumulating an initial store of facts for the development of a scientific theory; imagine the HMS Beagle's voyage curtailed and never delivering its collection of specimens.

References

[1] Sergey Bratus, Julian Bangert, Alexandar Gabrovsky, Anna Shubina, Daniel Bilar, and Michael E. Locasto. Composition patterns of hacking. In *Proceedings of the 1st International Workshop on Cyber Patterns*, pages 80–85, Abingdon, Oxfordshire, UK, July 2012.

[2] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: from buffer overflows to "weird machines" and theory of computation. *;login:*, December 2011.

[3] Thomas Dullien. Exploitation and state machines: Programming the "weird machine", revisited. In *Infiltrate Conference*, Apr 2011.

[4] Orin S. Kerr. Vagueness challenges to the Computer Fraud and Abuse Act. *Minnesota Law Review*, 2010.

[5] Alfredo Ortega and Gerardo Richarte. OpenBSD remote exploit. BlackHat 2007, June 2007.

[6] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications, 2009.

[7] Dan Rosenberg. Anatomy of a remote kernel exploit. `http://vulnfactory.org/research/defcon-remote.pdf`.

[8] Len Sassaman, Meredith L. Patterson, and Sergey Bratus. The science of insecurity. 28th Chaos Communication Congress, Dec 2011.

[9] Len Sassaman, Meredith L. Patterson, Sergey Bratus, Michael E. Locasto, and Anna Shubina. Security Applications of Formal Language Theory. Technical Report TR2011-709, Dartmouth College, 2011.