Exploit Programming

From Buffer Overflows to "Weird Machines" and Theory of Computation

SERGEY BRATUS, MICHAEL LOCASTO, MEREDITH L. PATTERSON, LEN SASSAMAN, AND ANNA SHUBINA

In memory of Len Sassaman, who articulated many of the following observations, connecting the mundane and the deeply theoretical aspects of hacking.

Sergey Bratus is a Research Assistant Professor of Computer Science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in mathematics from Northeastern University and worked at BBN Technologies on natural language processing research before coming to Dartmouth.
sergey@cs.dartmouth.edu

Michael E. Locasto is an assistant professor in the Computer Science Department at the University of Calgary. He seeks to understand why it seems difficult to build secure, trustworthy systems and how we can get better at it. He graduated magna cum laude from The College of New Jersey (TCNJ) with a BSc degree in computer science. Dr. Locasto also holds an MSc and PhD from Columbia University.
locasto@ucalgary.ca

Meredith L. Patterson is a software engineer at Red Lambda. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa, and has continued expanding the technique ever since. She lives in Brussels, Belgium.
mlp@thesmartpolitenerd.com

Len Sassaman was a PhD student in the COSIC research group at Katholieke Universiteit Leuven. His early work with the Cypherpunks on the Mixmaster anonymous remailer system and the Tor Project helped establish the field of anonymity research, and in 2009 he and Meredith Patterson began formalizing the foundations of language-theoretic security, which he was working on at the time of his death in July 2011. He was 31.

Anna Shubina chose "Privacy" as the topic of her doctoral thesis and was the operator of Dartmouth's Tor exit node when the Tor network had about 30 nodes total. She is currently a research associate at the Dartmouth Institute for Security, Technology, and Society, and manages the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research.
ashubina@cs.dartmouth.edu

Hacker-driven exploitation research has developed into a discipline of its own, concerned with practical exploration of how unexpected computational properties arise in actual multi-layered, multi-component computing systems, and of what these systems could and could not compute as a result. The staple of this research is describing unexpected (and unexpectedly powerful) computational models inside targeted systems, which turn a part of the target into a so-called "weird machine" programmable by the attacker via crafted inputs (a.k.a. "exploits"). Exploits came to be understood and written as programs for these "weird machines," and served as *constructive proofs* that a computation considered impossible could actually be performed by the targeted environment.

This research defined and fulfilled the need of such practical exploration in real systems that we must trust. Hacker research has also dominated this area, while academic analysis of the relevant computational phenomena lagged behind.

We show that at its current sophistication and complexity, exploitation research as a discipline has come full circle to the fundamental questions of computability and language theory. Moreover, application of language-theoretic and computation-theoretic methods in it has already borne impressive results, helping to discover and redefine computational models and weaknesses previously overlooked. We believe it is time to bring the hacker craft of finding and programming "weird machines" inside targets and the theorists' understanding of computational models together for the next step in designing secure, trustworthy computing systems.

The Rise of the Weird Machines

It is hard to say exactly when their story began; chances are that at the beginning they were thought of as just handy tricks to assist more important techniques rather than the essence of exploitation.

The classic "Smashing the Stack for Fun and Profit" by Aleph One [11] manages to explain the conversion of an implicit input data flow into altered program control flow in two short paragraphs:

So a buffer overflow allows us to change the return address of a function. In this way we can change the flow of execution of the program. ...
strcpy() will then copy [the shellcode] onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located. Once we reach the end of main and it tried to return it jumps to our code, and execs a shell.

For readers who concentrated on the details of constructing the shellcode (and encountered a hands-on exposition of syscalls and ABI for the first time) it was easy to miss the fact that both the implicit data flow and the subsequent transfer of control were performed by *the program's own code, borrowed by the exploit for its own purposes*. Yet it was this borrowed code, the copying loop of strcpy()and the function's post-amble that added up to the "remote execution" call as good as any API, into which the shellcode was fed.

This borrowing turned out to be crucial, far more important than the details of shellcode's binary instructions, as Solar Designer showed next year (1997): more of the target's code could be borrowed. In fact, enough code could be borrowed that there was no longer any need to bring *any* of your own executable code to drop a shell—the target process's runtime already conveniently included such code, in libc. One just needed to arrange the overwriting stack data the way that borrowed code expected it, faking a stack frame and giving control to the snippet inside libc's exec().

This was a handy technique for bypassing non-executable stack protections, and it was pigeonholed by many as such. But its real meaning was much deeper: the entire process's runtime address space contents were ripe for borrowing, as long as one spoke the language of implicit data flows (owing to the target's input handling logic flaws or features) that those borrowed pieces understood.

The borrowings did not need to be a one-off: they could be chained. Quoting all of non-code contents of Tim Newsham's 2000 post that probably holds the record for fewest words per idea value:

"Here's an overflow exploit [for the lpset bug in sol7 x86] that works on a non-exec stack on x86 boxes. It demonstrates how it is possible to thread together several libc calls. I have not seen any other exploits for x86 that have done this." [10]

It was soon generalized to any code snippets present in the target, unconstrained by the code's originally intended function or granularity. Borrowed pieces of code could be strung together, the hijacked control flow linking them powered by their own effects with the right crafted data arranged for each piece. Gerardo (gera) Richarte, presenting this technique, wrote less than half a year later: "Here I present a way to code any program, or almost any program, in a way such that it can be

fetched into a buffer overflow in a platform where the stack (and any other place in memory, but libc) is executable" [12].

So exploitation started to look like programming—with *crafted input data* for overflows or other memory corruptions—in really weird assembly-like instructions ("weird instructions") borrowed from the target. Nergal's "Advanced return-into-lib(c) Exploits" [9] described the chaining of faked overflow-delivered stack frames in detail, each borrowed post-amble with its RET instruction bringing the control flow back to the next faked frame, and out into the target's code or libraries, in careful stitches. Also, the granularity of features so stitched can be mixed-and-matched: should the load addresses of the desired snippets be obscured (e.g., with help of PaX hardening), then why not craft the call to the dynamic linker itself to resolve and even load the symbols, as is its job, let it do its thing, and then go back to snippet-stitching?

It does feel weird to so program with crafted data, but then actual assembled binary code is nothing but data to the CPUs in its fetch-decode-execute cycle, snippets of silicon circuits responsible for performing predictable actions when fed certain formatted inputs, then fetching more inputs. The exploit merely makes a "processor" out of the borrowed target code snippets, which implement the "weird instructions" just as digital logic implements conventional ones.

Altogether, they make up a "weird machine" inside the target on which the crafted-input program executes.

"Weird instructions" can be subtle, multi-step, and spread through the target's execution timeline. The original combination of strcpy() and a ret was a fair example, but just about any interface or library data interpretation code offers may offer a graceful specimen.

For example, Doug Lea's original memory allocator implementation keeps the freed blocks in a doubly linked list, realized as pointers in chunk headers interspersed with the chunks themselves. A bug in code writing to a heap-allocated buffer may result in a write past the end of the buffer's malloc-ed chunk, overwriting the next chunk's header with our crafted data. When the overwritten chunk is `free`-ed, the allocator's bookkeeping code will then traverse and patch the doubly linked list whose pointers we now control. This gives us a "weird MOV instruction" that takes four overwritten chunk header bytes and writes them where another four bytes are pointing!

This is beautiful, and we can program with it, if only we can cause the overwrite of a freed block and then cause the free() to happen. Such "weird instruction" techniques derived from a combination of an application-specific dynamically allocated buffer overwrite that corrupts the chunk headers and the normal malloc-ed chunk maintenance code are explained in detail in "Vudo malloc tricks" and "Once upon a free()" [7, 1].

Another famous example of a "weird instruction" is provided by the (in)famous printf-family format string vulnerabilities (in which the attacker could control the format string fed to aprintf()). From the computational point of view, any implementation of printf() must contain a parser for the format string, combined with an automaton that retrieves the argument variable's values from the stack and converts them to the appropriate string representations as specified by the %-expression. It was not commonly understood, however, that the `%n` specifier in format string caused that automaton to write the length of the output string printed so

far to the stack-fetched address—and therefore the attacker who controlled the format string and the quantity of output could write that length-of-output to some completely unanticipated address! (Even though printf was not passed a proper pointer to such a variable, it would grab whatever was on the stack at the offset that argument would be at, and use *that* as a pointer.)

What unites the printf's handling of the format string argument and an implementation of malloc? The "weird instruction" primitives they supply to exploits. This strange confluence is explained in "Advanced Doug Lea's malloc Exploits" [5], which follows the evolution of the format string-based "4-bytes-write-anything-anywhere" primitive in "Advances in Format String Exploitation" [14] to the malloc-based "almost arbitrary 4 bytes mirrored overwrite," for which the authors adopted a special "weird assembly" mnemonic `aa4bmo`.

Such primitives enable the writing of complex programs, as explained by Gerardo Richarte's "About Exploits Writing" [13]; Haroon Meer's "The(Nearly) Complete History of Memory Corruption" [8] gives a (nearly) complete timeline of memory corruption bugs used in exploitation.

Remarkably, weird machines can be elicited from quite complex algorithms such as the heap allocator, as Sotirov showed with his "heap feng shui" techniques [16]. The algorithm can be manipulated to place a chunk with a potential memory corruption next to another chunk with the object where corruption is desired. The resulting implicit data flow from the bug to the targeted object would seem "ephemeral" or improbable to the programmer, but can in fact be arranged by a careful sequence of allocation-causing inputs, which help instantiate the "latent" weird machine.

The recent presentation by Thomas Dullien (aka Halvar Flake) [3], subtitled *"Programming the 'Weird Machine,' Revisited,"* links the craft of exploitation at its best with the theoretical models of computation. He confirms the essence of exploit development as "setting up, instantiating, and programming the weird machine."

The language-theoretic approach we discuss later provides a deeper understanding of where to look for "weird instructions" and "weird machines"—but first we'll concentrate on what they are and what they tell about the nature of the target.

Exploitation and the Fundamental Questions of Computing

Computer security's core subjects of study—*trust* and *trustworthiness* in computing systems—involve practical questions such as "What execution paths can programs be trusted to not take under any circumstances, no matter what the inputs?" and "Which properties of inputs can a particular security system verify, and which are beyond its limits?" These ultimately lead to the principal questions of computer science since the times of Church and Turing: "What can a given machine compute?" and "What is computable?"

The old anecdote of the Good Times email virus hoax provides a continually repeated parable of all security knowledge. In the days of ASCII text-only email, the cognoscenti laughed while their newbie friends and relatives forwarded around the hoax warning of the woes to befall whoever reads the fateful message. We *knew* that an ASCII text could not possibly hijack an email client, let alone the rest of the computer. In a few years, however, the laugh was on us, courtesy of Microsoft's push for "e-commerce-friendly" HTMLized email with all sorts of MIME-enabled goodies, including "active" executable code. Suddenly, seriously giving "security"

advice to not "open" or "click" emails from "untrusted sources" was a lesser evil, all the sarcastic quotes in this paragraph notwithstanding. So long as our ideas met the computational reality we were dead right, and then those of us who missed the shift were embarrassingly wrong.

Successful exploitation is always evidence of *someone's* incorrect assumptions about the computational nature of the system—in hindsight, which is 20-20. The challenge of practical security research is to reliably predict, expose, and demonstrate such fallacies for common, everyday computing systems—that is, to develop a methodology for answering or at least exploring the above fundamental questions for these systems. This is what the so-called "attack papers" do.

What "Attack Papers" Are Really About

There is a growing disconnect between the academic and the practitioner sides of computer security research. On the practitioner side, so-called "attack papers"—which academics tend to misunderstand as *merely* documenting attacks on programs and environments—are the bread and butter of practical security (hacker) education, due to their insights into the targets' actual computational properties and architectures. On the academic side, however, the term "attack paper" has become something of a pejorative, implying a significant intellectual flaw, an incomplete or even marginal contribution.

However, a review of the often-quoted articles from *Phrack, Uninformed.org,* and similar sources reveals a pattern common to successful papers. These articles describe what amounts to an *execution model and mechanism* that is explicitly or implicitly present in the attacked environment—unbeknownst to most of its users or administrators. This mechanism may arise as an unforeseen consequence of the environment's design, or due to interactions with other programs and environments, or be inherent in its implementation.

Whatever the reasons, the point of the description is that the environment is capable of executing unforeseen computations (say, giving full shell control to the attacker, merely corrupting some data, or simply crashing) that can be reliably caused by attacker actions—essentially, programmed by the attacker in either the literal or a broader sense (creating the right state in the target, for example, by making it create enough threads or allocate and fill enough memory for a probabilistic exploitation step to succeed).

The attack then comes as a *constructive proof* that such unforeseen computations are indeed possible, and therefore as *evidence* that the target actually includes the described execution model (our use of "proof" and "evidence" aims to be rigorous). The proof is accomplished by presenting a computationally stronger automaton or machine than expected. Exploit programming has been a productive empirical study of these accidental or unanticipated machines and models and of the ways they emerge from bugs, composition, and cross-layer interactions.

Following [2], we distinguish between formal *proofs* and the forms of mathematical reasoning de-facto communicated, discussed, and checked as proofs by the community of practicing mathematicians. The authors observe that a long string of formal deductions is nearly useless for establishing believability in a theorem, no matter how important, until it can be condensed, communicated, and verified by the mathematical community. The authors of [2] extended this community

approach to validation of software—which, ironically, the hacker research community approaches rather closely in its modus operandi, as we will explain.

In other words, a hacker research article first describes a collection of the target's artifacts (including features, errors, and bugs) that make the target "programmable" for the attacker. These features serve as an equivalent of elementary instructions, such as assembly instructions, and together make up a *"weird machine"* somehow embedded in the target environment. The article then demonstrates the attack as a *program* for that machine—and we use the word *"machine"* here in the sense of a computational model, as in "Turing machine," and similar to "automaton" in "finite automaton."

Accordingly, the most appreciated part of the article is usually the demonstration of how the target's features and bugs can be combined into usable and convenient *programming primitives*, as discussed above. The attack itself comes almost as a natural afterthought to this mechanism description.

Exploits Are Working, Constructive Proofs of a "Weird Machine"'s Presence

It may come as a great surprise to academic security researchers that the practice of exploitation has provided an empirical exploration methodology—with strong formal implications.

For decades, hacker research on exploitation was seen by academia as at best a useful side-show of vulnerability specimens and ad hoc attack "hacks," but lacking in general models and of limited value to designers of defenses. The process of finding and exploiting vulnerabilities was seen as purely opportunistic; consequently, exploiting was not seen as a source of general insights about software or computing theory.

However, as we mentioned above, a more attentive examination of exploit structure and construction shows that they are *results akin to mathematical proofs*, and are used within the community in a similar pattern. Just like proofs, they are checked by peers and studied for technical "tricks" that made them possible; unlike most mathematical proofs, they are runnable, and are in a sense dual to correctness proofs for software such as the *seL4* project.

This proof's *syntactic* expression is typically a sequence of crafted inputs—colloquially known as the *"exploit,"* the same term used for the program/script that delivers these inputs—that reliably cause the target to perform a computation it is deemed incapable of (or does so with no less than a given probability). In some cases—arguably, the most interesting, and certainly enjoying a special place of respect among hackers—these crafted inputs are complemented with physical manipulations of the targeted computing environment, such as irradiating or otherwise "glitching" IC chips, or even controlling the values of the system's analog inputs.

The semantics of the exploit is that of a program for the target's computational environment in its entirety, i.e., the composition of all of its abstraction layers, such as algorithm, protocol, library, OS API, firmware, or hardware. This composition by definition includes any bugs in the implementation of these abstractions, and also any potential interactions between these implementations. The practical trustworthiness of a computer system is naturally a property of the composed

object: a secure server that relies on buggy libraries for its input processing is hardly trustworthy.

Constructing "Weird Machines"

From the methodological point of view, the process of constructing an exploit for a platform (common server or application software, an OS component, firmware, or other kind of program) consists of

1. identifying computational structures in the targeted platform that allow the attacker to affect the target's internal state via crafted inputs (e.g., by memory corruption);
2. distilling the effects of these structures on these inputs to tractable and isolatable *primitives*;
3. combining crafted inputs and primitives into *programs* to comprehensively manipulate the target computation.

The second and third steps are a well-understood craft, thanks to the historical work we described. The first step, however, requires further understanding.

Halvar Flake [3] speaks of the original platform's state explosion in the presence of bugs. The exploded set of states is thus the new, *actual* set of states of the target platform.

As much as the "weird machines" are a consequence of this state explosion, they are also defined by the set of reliably triggered transitions between these "weird" states. It is the combination of the two that make up the "weird machine" that is, conceptually, the substrate on which the exploit-program runs, and, at the same time, proves the existence of the said "weird machine." In academic terms, this is what the so-called "malicious computation" runs on.

From the formal language theory perspective, these transitions define the computational structure on this state space that is driven by the totality of the system's inputs, in turn determining which states are reachable by crafted inputs such as exploit-programs. The "weird machine" then is simply a concise description of the transition-based computational structures in this exploded space.

In this view, the exploitation primitives we have discussed provide the state transitions that are crucial for the connectivity of the "weird states" graph's components. In practice, the graph of states is so large that we study only these primitives, but it is the underlying state space that matters.

In a nutshell, it is only a comprehensive exploration of this space and transitions in it that can answer the fundamental question of computing trustworthiness: what the target can and cannot compute. The language-theoretic approach is a tool for study of this space and it may be the only hope of getting it right.

The Next Step: Security and Computability

The language-theoretic approach and "weird machines" meet at exploitation.

Practical exploration of real-world computing platforms has led to a discipline whose primary product is concise descriptions of unexpected computation models inherent in everyday systems. The existence of such a model is demonstrated by creating an exploit-program for the target system in the form of crafted inputs that cause the target to execute it.

This suggests that studying the target's computational behavior on all possible inputs as a language-theoretic phenomenon is the way forward for designing trustworthy systems: those that compute exactly what we believe, and do not compute what we believe they cannot. Starting at the root of the problem, exploits are programs for the *actual* machine—with all its weird machines—presented as input (which is what "crafted" stands for).

This approach was taken by Sassaman and Patterson in their recent research [15, 6]. They demonstrate that computational artifacts (which, in the above terms, make "weird machines") can be found by considering the target's input-processing routines as *recognizers* for the language of all of its valid or expected inputs.

To date, language-theoretic hierarchies of computational power, and the targets' language-theoretic properties, were largely viewed as orthogonal to security, their natural application assumed to be in compilation and programming language design. Sassaman and Patterson's work radically changes this, and demonstrates that theoretical results are a lot more relevant to security than previously thought.

Among the many problems where theory of computation and formal languages meets security, one of paramount importance to practical protocol design is algorithmically checking the *computational equivalence* of parsers for different classes of languages that components of distributed systems use to communicate. Without such equivalence, answering the above questions for distributed or indeed any composed systems becomes mired in undecidability right from the start. The key observation is that the problem is decidable up to a level of computational power required to parse the language, and becomes undecidable thereafter—that is, unlikely to yield to any amount of programmer effort.

This provides a mathematical explanation of why we need to rethink the famous "Postel's Principle"—which coincides with Dan Geer's reflections on the historical trend of security issues in Internet protocols [4].

References

[1] "Once upon a free()," *Phrack* 57:9: http://phrack.org/issues.html?issue=57&id=9.

[2] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. "Social Processes and Proofs of Theorems and Programs," technical report, Georgia Institute of Technology, Yale University, 1982: http://www.cs.yale.edu/publications/techreports/tr82.pdf.

[3] Thomas Dullien, "Exploitation and State Machines: Programming the 'Weird Machine,' Revisited," Infiltrate Conference presentation, April 2011: http://www.immunityinc.com/infiltrate/2011/presentations/Fundamentals_of_exploitation_revisited.pdf.

[4] Dan Geer, "Vulnerable Compliance," *;login: The USENIX Magazine*, vol. 35, no. 6, December 2010: http://db.usenix.org/publications/login/2010-12/pdfs/geer.pdf.

[5] jp, "Advanced Doug Lea's malloc Exploits," *Phrack* 61:6: http://phrack.org/issues.html?issue=61&id=6.

[6] Dan Kaminsky, Len Sassaman, and Meredith Patterson, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure," Black Hat USA, August 2009: http://www.cosic.esat.kuleuven.be/publications/article-1432.pdf.

[7]Michel "MaXX" Kaempf, "Vudo malloc Tricks," *Phrack* 57:8: http://phrack.org/issues.html?issue=57&id=8.

[8] Haroon Meer, "The (Almost) Complete History of Memory Corruption Attacks," Black Hat USA, August 2010.

[9] Nergal, "Advanced return-into-lib(c) Exploits: PaX Case Study," *Phrack* 58:4: http://phrack.org/issues.html?issue=58&id=4.

[10] Tim Newsham, "Non-exec Stack," *Bugtraq*, May 2000: http://seclists.org/bugtraq/2000/May/90 (pointed out by Dino Dai Zovi).

[11] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack* 49:14. http://phrack.org/issues.html?issue=49&id=14.

[12] Gerardo Richarte, "Re: Future of Buffer Overflows," *Bugtraq*, October 2000: http://seclists.org/bugtraq/2000/Nov/32.

[13] Gerardo Richarte, "About Exploits Writing," Core Security Technologies presentation, 2002.

[14] riq and gera, "Advances in Format String Exploitation," *Phrack* 59:7: http://phrack.org/issues.html?issue=59&id=7.

[15] Len Sassaman and Meredith L. Patterson, "Exploiting the Forest with Trees," Black Hat USA, August 2010.

[16] Alexander Sotirov, "Heap Feng Shui in JavaScript," Black Hat Europe, April 2007: http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.