The Bugs We Have to Kill

SERGEY BRATUS, MEREDITH L. PATTERSON, AND ANNA SHUBINA

Sergey Bratus is a Research Associate 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

Meredith L. Patterson is the founder of Upstanding Hackers. 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@upstandinghackers.com

Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society and maintains the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total.
ashubina@cs.dartmouth.edu

The code that parses inputs is the first and often the only protection for the rest of a program from malicious inputs. No programmer can afford to verify every implied condition on every line of code—even if this were possible to implement without slowing execution to a crawl. The parser is the part that is supposed to create a world for the rest of the program where all these implied conditions are true and need not be explicitly checked at every turn. Sadly, this is exactly where most parsers fail, and the rest of the program fails with them. In this article, we explain why parsers continue to be such a problem, as well as point to potential solutions that can kill large classes of bugs.

To do so, we are going to look at the problem from the computer science theory angle. Parsers, being input-consuming machines, are quite close to the theory's classic computing models, each one an input-consuming machine: finite automata, pushdown automata, and Turing machines. The latter is our principal model of general-purpose programming, the computing model with the ultimate power and flexibility. Yet this high-end power and flexibility come with a high price, which Alan Turing demonstrated (and to whose proof we owe our very model of general-purpose programming): our inability to predict, by any general static analysis algorithm, how programs for it will execute.

Yet most of our parsers are just a layer on top of this totally flexible computing model. It is not surprising, then, that without carefully limiting our parsers' design and code to much simpler models, we are left unable to prove these input-consuming machines secure. This is a powerful argument for making parsers and their input formats and protocols simpler, so that securing them does not require having to solve undecidable problems!

Parsers, Parsers Everywhere

To quote Koprowski and Binsztok [1]:

> Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation. For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

So do the lower layers of the network stack down to the IP and the link layer protocols, and also other OS parts such as the USB drivers (and even the hardware: turning PHY layer symbol streams into frames is parsing, too! [2]). For all of these core protocols, we add, their parsers have had a long history of failures, resulting in an Internet where any site, program, or system that receives untrusted input can be presumed compromised.

While we may believe in special programmers who write so-called critical software with the care and precision the rest of our tribe lacks, where are these secret coding schools training such ninjas? And if these programmers are so few and far between, can we really expect them to scale? Neither collective insanity nor collective negligence are comfortable to contemplate, but so we must as our reliance on software grows.

Perhaps we don't care nearly enough. After all, every C programmer experiences thousands of segfaults while learning the language and sees that the world doesn't collapse, nor does the computer suddenly become hostile. It certainly is annoying when programs crash, but it's easy enough to restart them—with an automatic watchdog, if need be. Indeed, few of us suspect how often embedded software in our devices gets restarted.

This habituation to crashes doesn't serve us well. It forms a false perception that "bugs are just bugs," and systems that engineer around them rather than fix them can be trustworthy, except in rare and exotic cases. But, in fact, this is where the common programming intuition lets us down badly.

A segfault is a would-be corruption of memory or state, an unexpected, out-of-type memory reference that got caught. It is eminently observable and doesn't result in much computation beyond the error. Therefore, it's easy to assume the same thing about any memory corruption—unless one is familiar with just how complete a programming environment a simple memory corruption can create for an attacker, and how far and wide beyond its expected execution paths a program can run after a memory corruption.

It's natural for programmers to view the executable binary generated from their programs through the prism of their source code. In that view, functions do not get jumped into sideways, nor are they called from locations other than their explicit call sites; variables retain their values unless assigned to by name or by reference; assembly instructions cannot spring into existence unless somehow implied by the code's semantics; and so on.

As attackers know, all of these expectations are false. In the gap between these expectations and the actual reality of binary execution at runtime, entire modes of programming sprang up. Around 2000, hacker researchers demonstrated that if one manages to overflow the program stack with what looks like a sequence of stack frames, one can construct arbitrary programs that will successfully execute in the corrupted process [3]. In 2007, an academic paper by Hovav Shacham [4] made this understanding precise by proving that a typical process is in fact a Turing-complete environment for such programming.

However, this kind of bare-boned exploit programming likely still feels too exotic for most programmers. Its power can only be experienced through practicing it, and most of us have our hands too full with the programming we need to do to pick up another, weirder kind of programming. So we'll need to approach it with a different set of intuitions, which are closer to the classic computer science than to hacking (although, as we will see, here hacking comes very close to the very foundations of computer science).

When Programs Crash, Where Do Their Proofs of Correctness Go?

C. A. R. Hoare developed the beginnings of the axiomatic proofs-of-correctness theory for programs in 1968. Owing to this theory, we see programs and their modules, functions, and constructs such as loops in terms of preconditions and postconditions, and chain these for proofs. Whenever such a chain can be constructed for the entire program, starting with its individual operations and statements, and the initial precondition is the atomic "True" (i.e., there are no additional preconditions), we say that we have proven the program's correctness (no matter what the inputs or the state of the rest of the world). Although few programmers actually end up proving their programs, generations of programmers have been taught to think of their loops and branches in terms of preconditions and postconditions. We intuitively understand the $P\{Q\}R$ notation even if we don't use it explicitly. That is, given preconditions P and code Q, postconditions R are assured.

But do we stop to think what happens when instead of P our code Q gets some $P' \neq P$? What will code Q be able to compute in that case? How far would possible conditions R' in $P'\{Q\}R'$ vary? Our intuition, based on axiomatic programming, does not tell us that—while an exploiter's intuition is all about it.

Some of our best theoretic means for achieving predictable code behavior, such as Proof-Carrying Code (PCC) and programming language safety guarantees, are of little help against the divergence in preconditions. For PCC, we can only be sure of what the code does if it's run within its specification [5]; otherwise, the proofs it carries do not preclude it from entering an unexpected "weird" state. The language-based guarantees rigorously proven on the source code can be broken either by the language's runtime implementation [6] or by compiler optimizations [7].

For parsers and the code that receives parsed input data, this question is even simpler: What happens when the inputs that hit the parser are invalid and unexpected? What will the parser itself compute then? If allowed through to the rest of the code, what effects will the inputs transformed by the parser have on it? Clearly, if the parser was supposed to reject the data and didn't, assumed preconditions to subsequent code on its path will not hold. The runtime world then belongs to whoever can predict the computational effects of violated preconditions, *even when the code is proven correct.*

It gets worse. Suppose we have a program that implements a simple finite state machine that responds to an input language. What happens when this code is fed inputs not in this language? Will the program still behave like a finite state machine, or will it present a much richer programming model to the attacker able to feed it custom-crafted inputs?

Accidentally Turing-Complete

The answer is almost certainly "yes." Software and even firmware intended as automata with limited, specialized purposes have been shown to actually play the role of a universally programmable Turing machine to attacker's inputs, which, for all their syntactic peculiarity, acted as programs for these machines. These inputs didn't even need to be malformed; either buffer or integer overflow bugs were similarly not a necessity.

For example, the standard ELF relocation code provided by the Linux dynamic linker and present in any dynamically linked process is driven by the relocation metadata present in every ELF executable. This code is meant to patch up the addresses in code that is loaded into a different address range than it was linked for—as a means of ASLR protection, for example, or simply because a previously loaded library already occupies part of the original address range—but it is capable of much more. In fact, craftily prepared well-formed metadata entries can make it carry out any computation at all, as if that code were a virtual machine and the relocation entries its bytecode [8]! This code was never meant nor written for such generality, but it can achieve it nevertheless [9].

What we think of as hardware is not far behind. For example, we trust the isolation of our processes to the x86 MMU, and we imagine it as a fairly simple mechanism that sets up our page tables on exec(), manages them on context switch, and translates every memory reference. Clearly, in this translation a finite automaton is involved, but in fact the MMU features are so rich that the configuration tables it interprets can be used to program anything—any Turing-complete computation [10]! Again, the MMU's logic was designed for a specific purpose, and great effort is spent on validating its correctness—but it turns out that it can do so much more than intended, with no bugs involved. Due to its feature-richness, merely well-formed crafted inputs suffice.

In short, computer security appears to have its very own parallel to Arthur Clarke's observation that "Any sufficiently advanced technology is indistinguishable from magic," namely, "Any sufficiently complex input format is indistinguishable from bytecode; the code receiving it is indistinguishable from a virtual machine."

The latter observation, of course, accords very well with the exploiters' everyday experiences. So long as the inputs are complex enough, and the software is correspondingly complex, there will be crashes, and some of these crashes will lead to full control of the receiving software.

The trick is putting these observations together and realizing what goes wrong. In full accordance with Clarke's laws, exploit developers lead in this exploration, because "The only way of discovering the limits of the possible is to venture a little way past them into the impossible." Indeed, in the programmers' mental models of their environments, exploits are supposed to be the impossible—and yet they exist.

The irony of these models is that the computational model of the general purpose computing, the Turing machine, was a proof of unsolvability, the impossibility of programming certain tasks due to the richness of the platform itself. The simplest of these is a particular kind of static analysis, a general algorithm for deciding statically whether a program would halt. The difficulty of this problem is by no means a fluke: according to Rice's Theorem, general algorithms for deciding other "non-trivial" properties of programs are in the same boat. This is not to say that static analysis of programs is hopeless but, rather, that it is hard, and this hardness is a matter of natural law that would not just yield to cleverness or extravagantly massive investment. As Geoffrey Pullum put it in his "Scooping the Loop Snooper" [11]:

> *No general procedure for bug checks will do.*
> Now, I won't just assert that, I'll prove it to you.
> I will prove that although you might work till you drop,
> you cannot tell if computation will stop.
>
> ...
>
> You can never find general mechanical means
> for predicting the acts of computing machines;
> it's something that cannot be done. So we users
> must find our own bugs. Our computers are losers!

This puts paid to the hope of exhaustively automating static security analysis for the kind of code that we most often write and use. Yet it is Turing's insights and his model of computing—an answer to Hilbert's tenth problem—that form the basis for most computers we know. Our software is just a layer on top of this totally flexible computer, and unless this software presents very simple parsers, that software is also likely to be totally flexible and cannot be proven to be secure—unless we programmers take great care to not use the full extent of this power and flexibility, and purposefully keep ourselves to simpler models that can be proven and verified.

Can We Verify Our Way Out of This Mess?

Maybe. First, we need to define the problem in a way that program verification tools can help. Then we need to pick a simple enough model of what parsing is and stick to it in our implementation.

Thus the long answer is, verification of parsers will help only if we co-design data formats and code that parses them. Parsers must create the preconditions for the rest of the proof; thus they should be the simplest machines possible, to ease effective verification. If you think this is a solved problem, it isn't. Quoting again from Koprowski and Binsztok,

> In the recent article about CompCert, an impressive project formally verifying a compiler for a large subset of C, the introduction starts with a question "Can you trust your compiler?" Nevertheless, the formal verification starts on the level of the [Abstract Syntax Tree] and does not concern the parser. Can you trust your parser?

So how simple should "simple" be?

Be Simple and Definite about What You Receive!

When software gets exploited by inputs—its execution takes a path it was never meant to take because of consuming the input data—it is obvious that the data is driving it to do so. But, in fact, although it may be less obvious, the data is driving the software even when it executes as expected. "The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program." [12]

This means that we should look at the data itself as a program—and model the parser code consuming it as an automaton driven by it. Then, so long as we keep this automaton simple, we can prove and verify its behavior on all possible inputs. We have the mathematics for it and a hierarchy of such automata by simplicity and power.

For example, consider a regular expression. We think of them as implemented by finite automata we can draw with circles and arrows, and emulate their execution by moving a coin from one circle to another along the arrow marked with the character we consume from the input string [13]. But then the string is what drives this automaton from state to state; it's the program for the automaton. The same is true for pushdown automata. It is obvious for a Turing machine: whatever goes on the tape is the program and is the input at the same time.

Regular expressions seem to be everyone's favorite way of validating inputs in scripting languages. This can be just right or can go horribly wrong, depending on the language of inputs one is trying to validate. Matching a regular language of inputs, one that consists of all strings matched by a regex anchored at the start and at the end of the string, would be just right. Of course, such languages work best for the data structures with no or limited nesting; for those like HTML or JSON that allow arbitrary nesting of their elements, it can go horribly wrong [14]. Validating arbitrarily nested HTML with regexes is a classic mistake, made by both novice Web developers and the designers of anti-XSS protections in IE 8 [15]. The mathematical reason for this world of XSS fail is simple: such languages are context-free or context-sensitive, and require at least a pushdown automaton to match them.

The purpose of the parser as a protector of the rest of the code is to match the correct inputs and drop the incorrect ones (without getting exploited itself, obviously). So we need to start by defining the language of the valid inputs, and then write the parser as the consuming automaton of the type we can validate. Usually this means keeping the input language regular or context-free, and using a regex (a finite state machine) or a pushdown automaton, respectively. We've seen how to safely approach what the parser consumes—but what about its outputs?

Types to the Rescue

To verify parsers, we need to first write their specifications. It's easy to say that parsers must consume strings, any strings, and reject those that are invalid or unexpected. But how can we describe what parsers must produce? What kinds of assumptions on input that passes the parser would be helpful for both ordinary programmers and the proof engineers seeking to verify their code?

This question goes back to the foundations of type theory. For example, the plight of the programmer who must rely on assumptions assured by the previous code was the subject of James Morris, Jr.'s "Types Are Not Sets" in 1973: "[The programmer] could begin each operation with a well-formedness check, but in many cases the cost would exceed that of the useful processing." Just as relevant to the programmers today as it was then!

The job of the parser then becomes clear once we see it from the type-theory angle. The parser eliminates strings; it introduces other objects of types that have to do with the program's semantics. The rest of the program assumes that these objects are well-typed; the parser is their constructor that builds them from the strings it consumes.

Parser bugs, then, generally come in two flavors: the parser code, instead of rejecting an invalid input, provides an attacker with a virtual execution engine for exploits, or the objects it constructs are not the type expected by the rest of the code. The former often occurs whenever the parser allocates and copies memory based on a value in user input it has neither fully parsed nor checked for consistency. Various integer overflows in X.509 and other ASN.1-based formats are examples of the latter: instead of the syntactically correctly encoded Bignum unbounded integer, the parser creates a bounded platform-default Fixnum [16]. So it is with Apache and Nginx chunked-encoding vulnerabilities, discussed later.

Format Foibles, Protocol Peeves

Exploiter intuition has long singled out certain syntactic features as the breeding grounds for parser vulnerabilities. Given the choice between constant-length and variable-length fields, the exploiter's money would be on the latter; several length fields that must agree for the message to be valid up the ante. A typical memory corruption scenario with such protocols involves copying some elements of input into buffers sized and dynamically allocated based on values supplied in the same input—and lying, to cause a buffer overflow. Although it's easy to blame such bugs on the implementers' negligence, it's undeniably the syntactic complexity of the underlying protocols that makes an implementer's mistake both more likely to happen and harder to catch.

Generally speaking, the more context a parser must keep to correctly parse the next element of the message, the more likely it is to get it wrong; the more complex the relationship between already parsed syntax elements and the remaining ones, the more likely an unchecked, unwarranted assumption is to slip through. Looking at the problem through the program proof lens, we can see the rapid accumulation of preconditions in context-sensitive protocols. However, the Internet—with its scale such that if a coding error can be made, it will be made—has a much more direct way of steering us towards regular and context-free formats.

In the Internet Protocol's early days, the variable-length IP options tacked on behind the constant-width IP header fields were considered essential. These days, their mere presence in a packet is enough for many firewall configurations to regard the packet as suspicious or to drop it outright. This happened for a good reason: IP option parsing bugs have plagued 1990s stacks (including firewalls like Raptor CVE-1999-0905 and Gauntlet CVE-1999-0683, which they caused to freeze or crash), made a few impressive appearances such as CVE-2005-0048 in the 2000s, and recently resurfaced as the "Darwin Nuke" kernel panic CVE-2015-1102 in Mac OS 10.10.2. Accordingly, the Internet de facto converged on the simpler constant-width IP header, a regular language—not by standard, but by a "rough consensus of firewalls."

Of course, any gains from this subsetting of IPv4 have been offset by the advent of IPv6 with its chains of variable-length Extension Headers, including nestable fragmentation headers. While concerned ASes filter and drop up to 40%(!) of certain kinds of IPv6 packets, newer RFCs call for limiting the allowed variations in header order and combinations [17]. This subsetting-by-firewall of IPv6 to a simpler grammar will likely continue.

The situation with the core trust infrastructure of the Internet, the X.509 PKI standard, is hardly more encouraging than that of IPv6. The wide variety of ways to represent basic data types such as integers and strings allowed under the ASN.1 Basic Encoding Rules (BER) makes parsing X.509 certificates and related data something of a guessing game as to what other implementations might mean; the "PKI Layer Cake" effort [15] revealed over 20 ways that different SSL/TLS implementations could interpret the same data in the certificate—including the Common Name! Thus a CA granting a certificate signing request for what looks like an innocuous domain could in fact create a certificate seen by the browsers as that of a different, high-value domain name. This abundance of differences is not surprising, since establishing equivalence of parsers is in fact a problem that becomes undecidable beyond a certain syntactic complexity, which X.509 significantly exceeds. Given the choice between ASN.1-based formats, the simpler DER and other encoding rules that fix respective canonical ways to represent each data type should be definitely preferred over BER, but syntactic complexity is the dark energy of the Internet: once created, it never goes away.

Speaking of SSL/TLS, the past year has been rich in famous SSL/TLS parser bugs. It wasn't just the infamous Heartbleed CVE-2014-0160; the GnuTLS Hello bug CVE-2014-3466 and Microsoft's Secure Channel bugs under CVE-2014-6321 demonstrate that the misery of complex input syntax really loves company.

While XML-based document formats are a definite improvement over the older binary ones, allowing a simple context-free subset to represent tree-like documents with recursively nested objects, the full XML specification still strays far enough from syntactic simplicity. Not surprisingly, the same elements, such as XML entities that introduce context-sensitivity to XML serve as a major source of its over 600 associated CVEs. By contrast, a simpler JSON, whose syntax would be context-free except for the requirement that its dictionary keys be unique, scores only about 60 CVEs; anecdotally, JSON parsers seem to be ahead of the game.

However, the Web has offset the simplicity that it promised in formats by an enormous explosion of computational power exposed to attacker inputs. Ubiquitous JavaScript ensures that the document one's client renders may have absolutely nothing to do with what one receives, precluding any kind of meaningful static analysis before rendering; instead of separating benign sheep from the malicious goats, the client has to put its trust into its sandbox being inescapable. And if this weren't bad enough, the combination of HTML5 and CSS in modern browsers already gives rise to programming models strong enough to exfiltrate one's passwords [18]. One may hope that such computations are accidental, but the demonstration that HTML and CSS3 are actually Turing-complete [19] leaves little hope that they will remain exotic or can be easily contained.

Chances are that we may need to rethink both the data formats and the computation models of the Internet before the mass of unwanted computation forces us into walled gardens of servers and peers somehow "trusted" not to poison our software.

Where Are We Now?

Decades of frustration have taught us to not roll our own crypto libraries. Although legacy crypto libraries are still complex and hard to use, new and simpler ones are just now emerging, like NaCl [20]. The Iron Age of crypto may be finally dawning on us.

With parsing, it's arguably worse. We are still in the Stone Age of parsing, despite a promising glint of Bronze and Iron here and there, or even an occasional laser beam. All across production programming, "rolling your own parser for speed" still reigns rather than raising skeptical eyebrows. Parser generators exist, but aren't seen as a vital necessity for input-handling code in either office document applications, messaging protocols, network stacks, or elsewhere; in short, their use cases are deemed limited rather than universal. Verified parsers are extremely rare; a majority of parsers are pwned-by-design, not least those we use in our cryptography.

One can continue blaming developers who don't "program securely" or fail to "validate inputs" (and some still do). However, a closer look at the nature of parser exploitation suggests this may be blaming the victim. Syntactically complex, context-sensitive protocols may in fact require the programmer to solve undecidable problems to create secure programs, an impossible feat.

As with all other kinds of engineering, the way forward lies in understanding which problems are impossible and which are merely hard, and not confusing the two. After all, every kind of engineering in the physical world works around its own impossibilities: conservation of energy and momentum, laws of thermodynamics, quantum-scale indeterminacy effects, and so on. Yet how sure can we be that random software engineers would so readily name the hard natural-law limits of their trade as physical engineers would?

It would be naive to expect that software engineering has no such limitations. Indeed, computability theory and complexity theory bring them to light. Nowhere do these limitations manifest themselves so cruelly as in our inability to predict computation. This inability is what we colloquially know as insecurity: we cannot trust our computers to stick to the computations we expect in the presence of inputs we don't control.

Building a Secure(r) Parser

We know the execution models for consuming inputs in which we can predict computation and protect it: these tend to be regular or context-free. We also know that context-sensitive and richer input languages harbor undecidable problems. As usual, the cure for an impossibility revealed by science is more science. In the case of parsers, we are lucky: we already have the mathematical models and the rough split of tasks into the possible and the impossible.

Our programming must follow these models and stay within the safe protocol designs that do not pose undecidable problems as requirements for "securing" them—that is, being able to automate testing of their implementations and reasoning about the possible courses their computations can take. For all the seeming flexibility and extensibility benefits of more complex protocols—and, respectively, more powerful computation models—building on them is like building on quicksand.

There is an important caveat for parsers explicitly hand-coded as finite automata, however: it should be clear from the code what kind of valid input any given part of it expects, and what syntactic construct it is responsible for parsing. For example, Nginx implements its parser of HTTP headers as a large hand-coded automaton (2300+ lines of C code, 57 switch statements, 272 single-character case statements). In 2013, it was found to incorrectly parse the chunk lengths in the HTTP chunked encoding (CVE-2013-2028), producing negative (signed) integers for large hexadecimal chunk lengths—exactly the same issue that was discovered for Apache in 2002 (CVE-2002-3092). It took over 11 years to find that bug in Nginx—and if you try looking through Nginx's ngx_http_parse.c to find where the chunk length is actually parsed, you will see why.

If the expected valid input is not intelligible from the code, finding bugs in it can take forever. In our experience, parsers whose code resembles the grammar of their expected inputs tend to do best. The Parser Combinator style of programming makes writing such code easy—and, although it was developed in functional languages such as Scala and Haskell, it's quite possible to use it in C/C++ and other languages as well. The Hammer parser construction kit is meant to demonstrate this; it requires no background in functional languages to use [21].

Help Me, Verifiable Parser, You Are My Only Hope!

When we look to the future of computers, what can we expect? Almost all kinds of programs will need to handle remote, untrusted inputs. The trend to connect everything and anything seems unstoppable; the "Internet of Things" and "cloud computing" (i.e., running trusted components of programs on remote systems) may only be its first wave.

Visions of self-driving cars, smart homes, and computerized medicine project from the current state of computing power, but not from its current trustworthiness. The only sustainable way to achieve these visions without an exploding attack surface is to make sure that all these programs exposed to hostile inputs can't be trivially exploited or disrupted by them. And if encrypted tunnels seem to be an answer, consider just how vulnerable the code base of our cryptographic infrastructure is to non-cryptographic attacks related to mere parsing of padding, PKCS message formats, and X.509 certificates.

The only hope for a secure connected future is software that can hold its own against the maliciously crafted inputs, without crutches such as firewalls, application proxies, antiviruses, and so on. This software will need to apply solid computation theory principles to what it accepts, and will accept only what it can validate. Once accepted, input can be turned into data types that will provide the rest of the software code with unambiguous preconditions. And although eliminating all bugs is provably impossible, the future should at least be free of the parser bugs on both input and output—the bugs we need to kill to build computers we can finally trust.

References

[1] Adam Koprowski and Henri Binsztok, "TRX: A Formally Verified Parser Interpreter," LMCS 2011.

[2] Travis Goodspeed, "Phantom Boundaries and Cross-Layer Illusions in 802.15.4 Digital Radio," First LangSec IEEE S&P Workshop, 2014: http://spw14.langsec.org/papers/8th-of-a-nybble.pdf.

[3] Gerardo Richarte, "Re: Future of Buffer Overflows," October 2000, Bugtraq: http://seclists.org/bugtraq/2000/Nov/32; Nergal, "Advanced Return-into-lib(c) Exploits: The PaX Case Study," *Phrack* 58:4, 2001; see also Sergey Bratus et al., "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," USENIX *;login:*, December 2011, for further history.

[4] Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," ACM CCS 2007.

[5] Julien Vanegue, "The Weird Machines in Proof-Carrying Code," First LangSec IEEE S&P Workshop, 2014: http://spw14.langsec.org/papers/jvanegue-pcc-wms.pdf.

[6] Eric Jaeger, Olivier Levillain, and Pierre Chifflier, "Mind Your Language(s): A Discussion about Languages and Security," First LangSec IEEE S&P Workshop, 2014: http://spw14.langsec.org/papers/MindYourLanguages.pdf.

[7] Vijay D'Silva, Mathias Payer, Dawn Song, "The Correctness-Security Gap in Compiler Optimization," Second LangSec IEEE S&P Workshop, 2015: http://spw15.langsec.org/papers/dsilva-gap.pdf.

[8] Shapiro et al., "'Weird Machines' in ELF: A Spotlight on the Underappreciated Metadata," USENIX WOOT 2013: http://www.cs.dartmouth.edu/~sergey/wm/woot13-shapiro.pdf.

[9] Mach-O and PE formats have comparable properties.

[10] Bangert et al., "The Page-Fault Weird Machine: Lessons in Instruction-less Computation," USENIX WOOT 2013: http://www.cs.dartmouth.edu/~sergey/wm/woot13-bangert.pdf.

[11] http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html.

[12] Taylor Hornby, quoted in Dan Geer, "Dark Matter: Driven by Data," Second LangSec IEEE S&P Workshop, 2015: http://spw15.langsec.org/geer.langsec.21v15.txt.

[13] This page's links explain how regex works in general, and particularly in Perl: http://perl.plover.com/Regex/.

[14] "Parsing HTML the Cthulhu Way": http://blog.codinghorror.com/parsing-html-the-cthulhu-way/, http://blog.codinghorror.com/content/images/2014/Apr/stack-overflow-regex-zalgo.png.

[15] http://p42.us/ie8xss/; see also Eduardo Vela Nava, David Lindsay, "Abusing Internet Explorer 8's XSS Filters": http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[16] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure": https://www.cosic.esat.kuleuven.be/publications/article-1432.pdf.

[17] See, e.g., F. Gont et al., "Observations on IPv6 EH Filtering in the Real World," 2015: https://tools.ietf.org/html/draft-gont-v6ops-ipv6-ehs-in-real-world-02.

[18] M. Heiderich et al., "Scriptless Attacks: Stealing the Pie without Touching the Sill," CCS 2012: https://www.hgi.rub.de/media/emma/veroeffentlichungen/2012/08/16/scriptlessAttacks-ccs2012.pdf.

[19] Eli Fox-Epstein, "Stupid Machines": https://github.com/elitheeli/stupid-machines.

[20] NaCl: http://nacl.cr.yp.to/.

[21] Hammer parser: https://github.com/UpstandingHackers/hammer. Also check out the Hammer Primer: https://github.com/sergeybratus/HammerPrimer for a gentle introduction.

Do you know about the USENIX Open Access Policy?

USENIX is the first computing association to offer free and open access to all of our conference proceedings and audio and video recordings. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your financial support plays a major role in making this endeavor successful.

Please help to us to sustain and grow our open access program. Donate to the USENIX Annual Fund, renew your membership, and ask your colleagues to join or renew today.

www.usenix.org/annual-fund