Editors: Patrick McDaniel, mcdaniel@cse.psu.edu | Sean W. Smith, sws@cs.dartmouth.edu

Beyond Planted Bugs in "Trusting Trust"

The Input-Processing Frontier

Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca "bx" Shapiro, and Anna Shubina

It's been nearly 30 years since Ken Thompson's "Reflections on Trusting Trust" lecture and its famous verdict that "You can't trust code that you did not totally create yourself."[1] If there is one practical lesson that the Internet has taught us since then, it's that you can't even trust your own code if it receives arbitrary inputs from the Internet. Sooner or later, a mixture of bugs or features turns connected code into an execution engine for hostile inputs—a *weird machine*.

Over time, exploitable bugs have become more complex and exploits more sophisticated; exploitation techniques first showed aspects of an art and then of a solid engineering process. However, all the bugs needed to compromise the software we use daily are likely already present in it. In this age of virtual hosting and cloud services, taking inputs from the Internet is just as dangerous as it was for the original Internet daemons. Despite defensive measures such as making program stacks nonexecutable and randomizing the target's address space, exploitable bugs migrate to other protocols or layers, while still giving attackers access to the defender's "crown jewels."

The 30 years that have passed since Thompson's speech is an eternity in Internet time and effort. When best efforts (in terms of both dedication and expenditure) at securing Internet-facing code are frustrated for so long, a deeper and yet not clearly understood principle must be at work. And, as is the case with most deep principles, it is likely hiding in plain sight.

Every Input Is a Program

Consuming input—any input—causes the consuming code and the underlying memory and processor to change state, typically on several levels of abstraction at once. In short, input drives the target through a computation. A program is as a program does—so *every input is in fact a program for its target*. It's a program in the same way that the input being matched to a regular expression is the program for the automaton underlying that RegEx implementation—the input drives the automaton through its states and transitions. It's also a program in the same sense that the content of a Turing machine's tape is a program for that machine as well as its input.

Information Is Instructions

We can conceive of information in two ways. First, we can rely on our common and traditional notion of information as some kind of inert data object, for example, a multimedia file. Our current biases assure us that surely this is the most inert type of data; after all, it's just *data* about pixels or sound waves, is it not?

Second, and much closer to objective reality, is the notion that all data is a stream of tokens of almost arbitrary complexity, and this stream of tokens is a *sequence of instructions* to the parser of its language. This sequence causes the parser to transition from state to state; read, write, copy, and allocate memory; and generally speaking, perform every kind of operation that a classic computational model such as a pushdown automaton or a Turing machine would. Therefore, we should speak not of code operating on input data but of *input data operating on code*—at least, on the part of the program that processes inputs.

Some inputs are very simple programs and cause very simple state changes. Regular expressions are quite manageable: we write them specifically to match inputs and ensure no states other than those of the regular expression automaton can be entered while matching. But the more complex the input and the more ad hoc the parser code, the less sure we can be of which states the code is capable of.

In other words, when presented with complex enough inputs and ad hoc code to handle them, we don't and can't fully know what kind of an automaton is inside the code and being programmed by inputs. Indeed, exploits are living, "pwning" proof that the induced computation can stray very far from the intended computation path—all the way to root shell.

1540-7993/14/$31.00 © 2014 IEEE Copublished by the IEEE Computer and Reliability Societies

The Message Is the Machine

Every valid discrete "message" (that is, a piece of information) passed between computers, network nodes, or program components (files, objects, function parameters, network packets, or frames) implicitly follows some grammar, which the code must implement. The tokens and constructs of the grammar drive the execution of the processing code's intended functionality.

But besides that programmer-intended functionality, there is also *latent* functionality that often holds far more power than the programmer intended, which can be triggered by a particular alignment of input tokens. This latent functionality may come from many sources, including emergent properties due to the composition of various code components and compiler- or runtime-inserted artifacts purporting to supply a full-featured execution environment.

The practical outcome is that programs often have access to much greater computational privilege than they need, and attackers are often able to find and expose this latent functionality.

As an example of emergent properties due to composition of components, consider the fact—long known to attackers—that almost all functionality needed to effectively exploit and control a remote target is already present on the target system, either in its OS or its libraries. A state-of-the-art exploit might chain up to six (the current Pwn2Own record) different bugs.[2] Long gone are the days when one stack overflow bug was enough to fully control the system. Native binary executable code in the payload has been rendered a rare luxury by the protective measures in general-purpose operating systems (but, alas, not in embedded and SCADA systems, where it remains a daunting reality).

Regarding compiler- or runtime-inserted artifacts supplying execution environments, consider exception handling in executables compiled with recent versions of GCC (GNU C compiler) using the DWARF format and logic to describe how to unwind the stack in case of an exception in particular functions. This logic involves a virtual machine (mapped into each process created from such an executable) and a bytecode that, for all practical purposes, is Turing-complete and can thus implement or hide Trojan executions.[3] Moreover, even the loading of a typical Linux ELF binary executable or library involves rewriting the binary's contents in memory, driven by the binary's metadata and the relocating code's complex logic. It turns out that this metadata alone is also Turing-complete with all the implications thereof (for example, detection of maliciously crafted metadata can be no better than heuristics looking for patterns of known malfeasance—the model that consistently fails against sophisticated attackers).[4]

The digital world has been conspicuously lagging in acknowledging the role of its own unsolvable problems.

From Untrusted Code to Untrusted Data

Thompson's caution that "No amount of source-level verification or scrutiny will protect you from using untrusted code"—and all the grand decidability theory behind it—might as well apply to *inputs* fed to ad hoc code. Because input can achieve full Turing-complete power with the help of the code that it drives, "verifying" input prior to its introduction into the system won't help. This spells doom for trust in any ad hoc, complex, and connected code, and this doom is on our doorstep.

The trick to restoring trust in code after these nearly 30 years turns out to be not just avoiding bugs, planted or accidental. It's about writing input-handling code in such a way that the effects of any inputs on it *can* be verified by examining the inputs (unlike programs, for which this is generally impossible). This means starving the input of its power to induce unexpected computation.

The input's power to induce computation is closely related to its power to exploit the handling code. Only very simple programs for very simple architectures submit to automatic reasoning about their effects. (For programming languages, such reasoning is called *verification*, a form of static analysis.) Hence, input-handling code and inputs, which are programs for this code, must be simple to allow such reasoning. Only then will we be able to trust the inputs—seen in their full power as programs—not to hijack the code. Formal language theory conveniently defines some such classes (as sketched in the sidebar).

Verification versus Validation

The previous discussion demonstrates a fundamental connection between what formal language theory calls *input validation* and *code verification*. Validation is the process of deciding whether input is as expected; verification is about proving that, granted the expectations, the code will have correct properties. Validation seems to be easy, but specifying expectations and picking properties can be hard, because general properties of code are impossible to verify or prove algorithmically. The

Exploits, Parsers, and Formal Languages

An exploit succeeds when it causes the target system to enter a state—on some level of programming abstraction—that was not expected by the target's original programmers. The best defense against crafted input exploits is to write input-handling code using programming models that are explicitly concerned with valid states and state transitions driven by input symbols.

Luckily, formal language theory provides just such models. It concerns itself with automata that consume input symbol by symbol, transition between well-defined states as they do so, and hand down a verdict on whether the consumed token sequence conforms to a valid language definition. Because all states and transitions are explicitly derived from the valid inputs' specification, and the actual implementation code can (and should) then be automatically generated from the same specification, there is a lot less chance of a programming error that would allow the system to be driven into an unexpected state by inputs.

In other words, an input parser that is a *recognizer* automaton for the valid inputs defined by a formal grammar has a much better chance of being secure against exploitation by crafted inputs.

Formal languages fall into broad classes by the computational strength of the automata required to recognize them.

Regular languages—familiar to the reader as those that can be precisely matched by a regular expression—require only a limited amount of state, regardless of the input string's length. Such automata can be implemented without resorting to dynamic memory allocation, and thus without risking any of the nasty overflow associated with malloc()-ing and copying untrusted data.

The next input language complexity class, *context-free languages*, allows arbitrary recursion depths of embedding data structure representations but draws the line at interdependence of sibling elements. Such parsers still tend to be manageable to verify and maintain; they also fulfill the appetite for transmitting structures that can be recursively nested, such as JSON structures in Web apps.

Harder to parse—and for auditors of parsers to verify—are *context-sensitive* and recursively enumerable languages. In context-sensitive languages, validity of nested object representations depends on a host of other objects processed either before or after the current object representation is parsed. This property often leads to assumptions that aren't verified by the parser and blow up in subsequent processing code, leading to memory corruption and exploitable bugs. Notorious in this respect are message formats that contain multiple object length fields that must agree across the whole message.

Input-handling code is inherently risky. To offset this risk, developers must rigorously design valid inputs and adopt the programming practices to match. In particular, defenders' tasks can be made substantially more tractable by using simpler language formats that don't exceed regular or context-free language strengths. Adhering to these formats allows for simpler, less state-hungry, and therefore more likely bug-free parsers.

Conversely, allowing more complex input formats paints defenders into a corner where they must solve hard or in fact unsolvable problems. Attempting to find 80/20 engineering solutions to these problems is the fallacy that is the underlying cause of the current Internet insecurity epidemic (http://langsec.org/papers/Sassaman.pdf).

For a more in-depth synopsis of exploits, parsers, and formal languages, see http://doi.ieeecomputersociety.org/10.1109/MSP.2014.1.

underlying fundamental computer science result is known as the halting problem or its equivalent, Rice's theorem, which states that there is no general and effective method to decide whether a given program terminates or has any other nontrivial property (such as whether a program always computes the square of the number it inputs). A quandary arises.

Treating input as a program—as exploitation does—leads us out of this quandary. We ask, can we *verify* inputs as programs in terms of their effects on the target? The problem seems harder, but solving it is necessary to deny exploitation by input. The only answer, then, is to keep the input language simple enough—say, regular expression strength, or deterministic context-free (equivalently, deterministic pushdown automaton) strength if recursive nesting of data structures in the input is necessary—and to write the code that validates it accordingly. Then, the code will be verified, and the input will be validated by that code, without fear of extra states and runaway computation.

Destructive Disagreements

A common source of trust failures is the different dialects of message formats—from network packets to package files—that are mutually misunderstood by communicating programs or layers. Such distinct dialects offer attackers the opportunity to craft messages that peers will understand differently. The consequences are devastating for any trust or security assumptions that rely on the correct and *coordinated* perception and processing of these messages.

Simply put, whenever two input parsers are involved, a disagreement between them about an input might destroy trust, even if both parsers accept the input safely.

This effect is particularly devastating if certificates or signatures are involved. The disagreeing parsers might reside on different systems, as was the case with X.509 certificate authorities (CAs) and browsers that saw different domain information in the certificate signing request (CSR) and the signed certificate, respectively. The CA's parser

interpreted the CSR to contain an innocent domain name belonging to the requester and signed it, whereas the browser's SSL client interpreted the same data to be a high-value domain name belonging to another entity.[5]

Alternatively, the parsers might reside on the same system as parts of a binary tool chain, such as the package signature verifier and the package installer in the case of the Android Master Key type bugs.[6] The bugs featured a Java library cryptographic signature verifier and a C++ installer, both of which interpreted the compressed archive—but disagreed regarding its contents. As a result, unsigned content could be installed.

This problem is potentially present in chains of trust wherever both the signature and the signed object are contained in packages with nontrivial packaging formats. Their respective locations inside the package are computed from the package metadata; thus, the correctness of signature verification depends on the correctness and agreement of metadata interpretation by all components. (Besides the already mentioned examples of X.509 and Android Master Key bugs, see the classic intrusion-detection system evasion research.[8,9])

The kinds of messages (programs) that can be algorithmically decided to cause equivalent computation must be even simpler than the programs for which we can decide whether they halt. Thus, the message formats that we want to ensure are parsed the same on different parsers must be simple enough as a language, and the respective parsing code must match that simplicity exactly.

There Can Be No Chain of Trust in Babel

A trust chain is in fact a chain of parsers that interpret binary content to prevent unexpected computation throughout the execution chain. It's entirely natural to break up cryptographic verification into modules or even separate tools—after all, this is what Unix's philosophy of small tools doing one thing well encourages.

However, when these parsers disagree, a Babel-like explosion of diverging interpretations and parser-specific dialects becomes a danger to signing schemes, object serialization, and even security proof infrastructures. To paraphrase a well-known line from *The Matrix*, "What good is a signature, Mr. Anderson, if you can't really see the document?"

Metadata Malicious, Mutable

Because automatic reasoning about code is generally hard, we simply sign code and later check signatures to convince ourselves that it hasn't changed since signed by someone we trust. However, this ignores the engineering reality that the code will be rewritten and combined with other modules, which might completely change the properties of the overall program image.

As software engineering gets more complex (Remember statically compiled executables? Try finding any on your system!), so do transformations of binary code and data. For example, relocation of binary code used to mean patching absolute addresses in it to account for loading the code at a different address than linked for. Now, there are more than a dozen types of relocations, and the GNU/Linux code that applies them resembles a virtual machine's implementation of a bytecode. On Mac OS X, relocation entries *are* bytecode designed to be executed by a virtual machine. Perfectly well-formed relocation entries are in fact Turing-complete in a standard ELF-based GNU/Linux environment,[4] and the same is likely true for Mach-O and Portable Executable formats.

Perhaps more surprising is the x86 address translation mechanism that composes physical memory frames into the abstraction of a virtual address space. Its logic—fed by page tables, interrupt descriptors (IDTs), memory segment descriptors (GDTs), and 32-bit hardware task-switching descriptors—turns out to be Turing-complete![7]

All these "tables" turn out to be programs for their respective interpreter logic (software or hardware), capable of arbitrarily transforming the signed code supposedly "frozen" in a trusted state. Unless all these kinds of "table" metadata are watched and can be effectively reasoned about, the transformed code can't be trusted.

As before, this means that software engineering metadata that goes into composing multiple pieces of code into a single runtime image must stick to the simplest possible formats—or be treated as code, with their immutability assured with strong cryptography and unambiguous ways of locating them and their signatures. This sounds a bit like the chicken-and-egg problem, does it not? Simplifying the data and its respective parsers to verifiable strengths suddenly sounds like a better deal for trust chains.

In the physical world, engineering is based on the firm understanding of unsolvable problems, rendered such by fundamental laws, such as conservation laws, that we know can't be bent by cleverness or hard work and funding. The digital world has been conspicuously lagging in acknowledging the role of its own unsolvable problems. Public perception still regards computerization as magic that can significantly improve any human endeavor when applied with sufficient zeal. Yet, symbolic manipulations are subject to natural limitations as harsh as physical ones.

Ubiquitous insecurity of connected systems and spectacular failures of large-scale integration projects are early cautionary examples of how the digital utopia fails.

Will we ever be able to trust connected computers? Can we pull out maliciously crafted inputs' poison teeth? At the very least, we must rethink the dominant design attitudes that got us here, such as the idea that document viewers should "fix" erroneous input rather than discard it out of hand as well as the notion of extending document formats until documents require Turing-complete interpreters to render. The same goes for the designs that require scripts in general-purpose programming languages to be executed before users can even begin to judge a document's provenance.

The effective trust model of designs that ignore inherent computing limitations is the "leap of faith," ending in expensive subscription-based heuristic Band-Aids or in blaming users—that is to say, victims. Worse yet, large-scale deployment of fragile, untrustworthy software creates vulnerability to direct physical damage. The only winning move is not to play. ■

Acknowledgments

We thank Rik Farrow, whose help in getting the LangSec message out has been invaluable.

References

1. K. Thompson, "Reflections on Trusting Trust," *Comm. ACM*, vol. 8, no. 27, 1984, pp. 761–763.
2. "A Tale of Two Pwnies (Part 1)," Chromium blog, 22 May 2012; http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.
3. J. Oakley and S. Bratus, "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code," *Proc. 5th Usenix Conf. Offensive Technologies*, Usenix, 2011; https://www.usenix.org/event/woot11/tech/final_files/Oakley.pdf.
4. R. Shapiro, S. Bratus, and S.W. Smith, "'Weird Machines' in ELF: A Spotlight on the Underappreciated Metadata," *Proc. 7th Usenix Conf. Offensive Technologies*, Usenix, 2013; https://www.usenix.org/conference/woot13/workshop-program/presentation/Shapiro.
5. D. Kaminsky, L. Sassaman, and M. Patterson, "PKI Layer Cake: New Collision Attacks against the Global X.509 CA Infrastructure," 5 Aug. 2009; http://ioactive.com/pdfs/PKILayerCake.pdf.
6. J. Freeman, "Android Bug Superior to Master Key," The Realm of the Avatar blog, http://www.saurik.com/id/18.
7. J. Bangert et al., "The Page-Fault Weird Machine: Lessons in Instruction-Less Computation," *Proc. 7th Usenix Conf. Offensive Technologies*, Usenix, 2013; https://www.usenix.org/conference/woot13/workshop-program/presentation/Bangert.
8. M. Handley, V. Paxson, and C. Kriebich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," *Proc. 10th Conf. Usenix Security Symp.*, vol. 10, 2001; https://www.usenix.org/conference/10th-usenix-security-symposium/network-intrusion-detection-evasion-traffic-normalization.
9. T. Ptacek and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Networking Intrustion Detection," 1998; http://citeseerx.ist.psu.edu.viewdoc/summary?doi=10.1.1.119.399.

Sergey Bratus is a research assistant professor of computer science at Dartmouth College. Contact him at sergey@cs.dartmouth.edu.

Trey Darley is a senior security strategist with Splunk's Security Practice. Contact him at trey@treyka.net.

Michael E. Locasto is an assistant professor at the University of Calgary. Contact him at locasto@ucalgary.ca.

Meredith L. Patterson is the founder of Upstanding Hackers. Contact her at mlp@upstandinghackers.com.

Rebecca "bx" Shapiro is a PhD student at Dartmouth University. Contact her at bx@cs.dartmouth.edu.

Anna Shubina is a research associate at the Dartmouth Institute for Security, Technology, and Society. Contact her at ashubina@cs.dartmouth.edu.