Beyond Planted Bugs in "Trusting Trust": the Input-processing Frontier

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

DRAFT

It's been nearly thirty years from Ken Thompson's "Reflections on Trusting Trust" and its famous verdict that "You can't trust code that you did not totally create yourself." If there is one practical lesson that the Internet taught us since then, it is that one cannot even trust one's own code so long as that code meets arbitrary inputs from the Internet. Sooner or later a mixture of bugs or features turns the connected code into an execution engine for hostile inputs – indeed, it was sooner rather than later for the original Internet daemons.

Over time, exploitable bugs became more complex, exploit payloads more sophisticated. Their composition first showed aspects of an art and then of a solid engineering process. Still, with a few exceptions, code connected to the Internet cannot be trusted.

Even though everything Thompson predicted, including well-placed microcode bugs, has come to pass, there seems to be no need for a malicious entity to insert bugs into the software most of us use daily on the Internet. The input-subvertible bugs are already there. When a mechanism used for subversion is dealt with (as happened to the executable stack and predictable uniform address space layout), bugs simply seem to migrate to another protocol or layer. Input is still just as dangerous as it was for early implementations of SMTP and DNS.

1 Every input is a program

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, think of a multimedia file – our current biases assure us that surely this is the most inert type of data there can be; after all, it is *data* about pixels or sound waves, is it not?). Second, and in fact much closer to objective reality, is that all data is a stream of tokens of almost arbitrary complexity, and this stream of tokens is actually a *sequence of instructions* to the parser of that language. We therefore get the maxim of data operating on code — not code operating on data!

There is a deeply theoretical reason for this: *every input is in fact a program for its target.* It's a program in the same sense as the input being matched to a Regular Expression is the program for the automaton underlying the implementation of that RegEx: the input drives the automaton through its states and transitions. It's also a program in the same sense as the content of a Turing machine's tape is a program for that machine as well as its input. 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.

Some inputs are very simple programs and cause very simple changes of state. Regular expressions are quite manageable: we write them specifically to match inputs and make sure no other states 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 what states the code is capable of.

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

The message is the machine. Perhaps a more general way of stating the relationship between information and code is that every discrete "message" passed between computers, network nodes, or program elements or components (files, objects, function parameters, network packets or frames) implicitly follows some grammar. The tokens and constructs of the grammar drive the execution of the intended functionality present in the processing code.

But besides that programmer-intended functionality, there is also *latent* functionality that often holds far more power than the programmer either intended or understands, and it can be triggered by particular alignment of input tokens. This latent functionality exists due to a number of sources. Two such sources among many others are (1) emergent properties due to the composition of various code components and (2) compiler- or runtime-inserted artifacts aimed at supplying a "full–featured" execution environment.

The practical outcome is that programs often have access to much greater computational privilege than they need, and an attacker is often able to find and expose this latent functionality. Once the attacker succeeds, there go any security perceptions, testing outcomes, or trustworthiness assumptions a program's owner might hold.

From untrusted code to untrusted data. Thus 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) may as well apply to *inputs* fed to ad-hoc code. Since input may in fact achieve full Turing-complete power with the help of the code that it drives, no amount of "verifying" the input prior to it entering the system may help. This spells doom for trust in any code that is ad-hoc, complex, and connected. This doom is apparently on our doorstep.

Coming back to Ken Thompson's "Reflections on Trusting Trust", the trick to restoring trust in code after these nearly thirty 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 it's generally impossible). This means starving the input of its power to induce computation in, 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 both be simple in order to allow such reasoning. Only then we'll be able to fully trust the inputs as programs not to hijack the code. Formal language theory conveniently defines some such classes.

Verification vs. Validation. This creates an fundamental connection between what theory calls *input validation* and *code verification*. Validation is the process to decide 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 too general propeties of code are known to be impossible to algorithmically prove or verify. 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 language of inputs to a simple enough model — say, to regular expression strength, or to deterministic context free (equivalently, deterministic pushdown automaton) strength if recursive nesting of data structures in the input is needed – 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 of the eloping computation.

Bob is bytecode, too Cryptography has a special place in trust, but systems that use it must also deal with parsing. Depending on the cryptographic protocol and network transport, some of this parsing must happen before the actual origin of the message is assured. Thus Bob, or, rather, Mallory, may try to drive the parser with crafted data, just as any other parser. In other words, even Bob might be bytecode.

2 Beyond bugs in "trusting trust"

Bugs resulting from unconstrained parsing or incomplete and unsound recognition of data streams are the most familiar way for software to be untrustworthy, but they are not the only one. Even if we somehow eliminated all memory corruption bugs in individual input handlers, significant issues will remain. Not surprisingly, they are also related to computation and recognition.

Destructive disagreements. Simply put, whenever two input parsers are involved, a disagreement between them about an input may destroy trust, although both parsers accept it safely.

This effect is particularly devastating if certificates or signatures are involved. The disagreeing parsers may reside on different systems, as was the case with X.509 Certificate Authorities and browsers that saw different domain information in the Certificate Signing Request and the signed certificate respectively. In that example, the CA's parser interpreted the CSR to contain an innocent domain name and signed it, whereas the browser's SSL client interpreted the same data to be a high value domain name.

Alternatively, the parsers may reside on the same system as parts of a binary toolchain, such as the package signature verifier and the package installer in the case of the recent Android Master Key bugs. The latter 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 signatures of objects must be checked. For software engineering reasons, both the signature and the signed object tend to be contained in packages with non-trivial packaging formats. Their respective locations inside the

package are computed from the package metadata – and thus correctness of signature verification depends on the correctness and agreement of metadata interpretation by all components.

It's entirely natural to break out cryptographic verification into a module or even a tool separate from other package-related operations; the Unix philosophy of small tools doing one thing well may even encourage this. Still, separate modules mean separate parsers, and the danger of them diverging enough to break trust. "What good is a signature, Mr. Anderson, if you can't really see the document?"

Metadata malicious, mutable. Since 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 can trust to not tamper with it. Still, this ignores the engineering reality that the code will actually be re-written and combined with other modules, which may 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![1]), so do transformations of binary code and data. For example, relocation of binary code used to mean patching of absolute addresses in it to account for loading the code at a different address than compiled for. Now there are over a dozen types of relocations, and the GNU/Linux code that applies them closely resembles a VM's implementation of a bytecode. On Mac OS X, relocation entries *are* bytecode, designed to be executed by a virtual machine. In gets better – perfectly well-formed relocation entries are in fact Turing-complete in a standard ELF-based GNU/Linux environment, and the same is likely true for Mach-O and PE formats.

Unexpected, powerful execution engines lurk in other standard environments. DWARF-based exception handling also turns out to be Turing-complete. 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 (IDT), memory segment descriptors (GDT) and 32-bit hardware task switching descriptors turns out to be Turing-complete!

All of these "tables"turn out to be programs for their respective interpreter logic (software or hardware), capable of arbitrarily transforming supposed 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 hardly be trusted.

This, as before, 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 else themselves be treated as code, and their immutability assured with strong cryptography and unambiguous ways of locating them and their signatures. Sounds a bit like chicken-and-egg problem, does is not? Simplifying these data and their respective parsers to verifiable strengths suddenly sounds like a better deal for trust chains.

The packet is the program. A common source of trust failures resides in the dialects of message formats, from network packets to package files, that are mutually mis–understood by communicating programs or layers. Such mutually distinct dialects offer the opportunity to craft messages that peers will understand differently. The consequences are often devastating for any trust or security assumptions that rely upon the correct *and coordinated* perception and processing of these messages.

Interpretation is essentially a computation driven by the message. As bits of a message are consumed by the code that handles them, computation caused in that code produces whatever

[1] OpenSolaris simply removed the static compilation option years ago `https://blogs.oracle.com/quenelle/entry/solaris_10_ack_where_did`, and Linux distributions such as RedHat/Fedora discourage it.

subsequently passes for the message's meaning. Although different computations may produce the same result, so far as the consumer of that result is concerned, the safest way to be sure that two systems interpret a message equivalently is to require that the computations caused by this message on these systems are equivalent.

For messages that are programs in a Turing-complete language, the problem of checking equivalence is as hard as halting problem. Such messages, a.k.a. programs that can be proved by an algorithm to cause equivalent computation, must in fact be even simpler than the programs about which we can decide whether they halt or not. Thus the message format that we want to assuredly parse the same on different parsers must be simple enough as a language; and so must be its respective handling code.

There can be no chain-of-trust in Babel. By definition of a chain, we expect the chain of trust to consist of several links, each a separate piece of hardware or software. Still, they are to see through the trustworthy execution of the same binary content, even though they have different views of it, at different depths of interpretation.

When the format of the trusted content allows for ambiguity or implicit assumptions, a Babel-like explosion of dialects and interpretations becomes a danger. Once that happens, the links of the trust chain may no longer agree on what content they safeguard, leading to breaches. There can be no chain of trust in Babel.

This potential for divergent recognition arises in most input processing activities far beyond the exchange of packets and packages: object serialization, command sequences passed through `ioctl`'s, security–processing of event token streams such as system call sequences, or intrusion alerts (i.e., SIEM), the parsing and interpretation of input to security proofs or proof infrastructures, etc. A particularly interesting area is the AI area of "malicious argumentation", where the goal is for an adversary to offer specious but well–formed arguments that cause the victim to take an incorrect or sub-optimal decision.

3 So can we fix it?