Trusting Trust on Mars: Thoughts on Urbit Security

N. E. Davis ~lagrev-nocfep
Urbit Foundation

Abstract

Via Ken Thompson's classic reflections on compiler safety and the injection of "undetectable" and irreversible malicious bugs into binaries, we analyze aspects of the security of Urbit as a two-level executable system, i.e. as a virtualized machine running on an execution layer.

Contents

Urbit Systems Technical Journal I:2 (2024): 55–65.

Address author correspondence to ~lagrev-nocfep.

1 Introduction

Ken Thompson's famous 1983 Turing Award lecture, "Reflections on Trusting Trust" (Thompson, 1984), concisely illustrated a fundamental difficulty with reposing confidence in any software stack one has not built oneself from assembler code. He postulated a compiler which had been modified to match a particular pattern and inject malicious code (such as a login backdoor or privilege escalation) into any program it produced; and a more insidious scenario in which the compiler knows to change its own source upon compilation to perpetuate the backdoor attack forwards. His exposition proceeded from the production of a quine, to a learning compiler, to the malicious "trusting trust" attack. The (non)solution Thompson proposed as epitome, "you can't trust code that you did not totally create yourself," was cold comfort to the computer security expert.[1]

Each component of the trusting trust attack can be demonstrated in Nock or Hoon and to some extent realized in Arvo. Countermeasures and implications for the overall security of the Urbit stack are also considered.

2 Nock Quines

A quine is an example of a code program that produces its own source; it is a fixed point of its syntax and execution. Some languages make this relatively straightforward, particularly those with rich string syntax. As Wikipedia states, "quines are possible in any Turing-complete programming language, as a direct consequence of Kleene's recursion theorem"; an older version of the same page emphasizes the utility of the ability to output a computable string.

Nock operates by applying a formula against a subject. The subject–formula pair is reduced by the first matching pattern from the left. A strict quine has no input, however, for Nock

[1] Indeed, Forth has been employed as a clean bootstrap platform since it can be implemented straightforwardly in machine language on many target embedded architectures. This is more for pragmatism than security, but the possibility of employing this as a strict security measure exists.

a formula is a partial function that must be evaluated against a subject. We must reconcile the subject as a context but not an input, as it were. A proper Nock quine therefore includes both, and replication of the subject–formula pair should be a requirement for a true quine.

The trivial Nock quine is simply an address replication using the Nock 0 operator.[2]

```
[[0 1] 0 1] [[0 1] 0 1]
```

Indeed, this suggests a family of quines built on tree structure replication; `~pinhul-radlyr` suggested the following:[3]

```
[[[0 1] [[0 2] [0 3]]] [[0 1] [[0 2] [0 3]]]]
```

and the recursive generalization is straightforward. Following Thompson, we consider such a quine to be a "compiler" in that it produces a program from a program.[4]

3 A Learning Compiler

Thompson proceeds from the notion of a quine to constructing a learning compiler, or a compiler that can modify the source of its successor in such a way as to extend the semantics of the language.

Although raw Nock is extremely uncommon in practice on Urbit, it is slightly more common to manipulate vase-mode values of Hoon and rather frequent to modify the Hoon compiler itself (at least in kernel development). A modified Hoon compiler (let us call it "+(Hoon)") is capable of perpetuating language changes forward into the nouns resulting from source. A simple example is adding an additional rune or altering the behavior of the parser (in `++vast`) or AST Nock compiler (in `++mint:ut`). Given access to one's `%base` desk, the possible modifications are endless (Listing 1).

[2] I am indebted to `~dozreg-toplud` for first bringing this simple quine to my attention.

[3] `https://github.com/jpt4/icfp2024code/blob/main/quine.rkt`

[4] Further articles of interest on producing quines have been provided by David Madore and Eric Hokanson.

Listing 1: A simple example of pattern-detecting adaptive code in Hoon and Nock.

```
::  Hoon
?:(=(- [2 3 4]) [2 3 5] -)
::  Nock
[6 [5 [0 2] 1 2 3 4] [1 2 3 5] 0 2]
```

Given control of a compiler, Thompson then proceeds to spring the trap: "a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched" (p. 763). That is, all code descended from the compromised compiler in his example may be considered contaminated by a self-perpetuating Trojan horse.

Urbit does not directly distribute Nock code as executable nouns today. Any practical attacks based on Nock would require distributing the malicious Nock code as data or inducing a Hoon compiler to produce the malicious Nock code inadvertently. Such compromised code would then have to executed against an appropriate subject to produce the desired malicious consequence.[5]

Critically for Urbit, the Hoon compiler at any one state of the kernel is used to produce the next state. "Live" code is not sent over the network: one sends source files which are then compiled locally into executable Nock nouns. Given such a compromised compiler, "no amount of source-level verification or scrutiny will protect you from using untrusted code" (*ibid.*). The trusting trust attack is complete and the system has been fundamentally compromised.

Another relatively unexplored avenue is to have Nock formula manipulate the rules themselves. Typically, Nock rules are treated as constants, and Hoon code that utilizes them often simply specifies them as such for legibility. This is not necessary, however, and changing the rules this way is straightforward in raw Nock, if a little difficult to conceptualize (a Hoon

[5] Since a Nock formula is a partial function, Hoon should be considered as a hypotactic language. That is, each expression is a phrase subordinated to its subject. Crossover manipulations between the subject and formula are thus possible, as long as an iterated cycle of using a formula against a subject to produce a new subject is followed.

example is provided in Listing 2). (This is salient in iterated cycles, a subject–formula pair yielding the next subject for subsequent formula evaluation.)

Listing 2: Hoon which can modify the rules of supplied Nock formulae arbitrarily. For simplicity, lists are used when in practice it would be done in cells and tuples.

```
=/  a=(list @)  [4 1 1 ~]
=|  b=(list @)
|-  ?~  a  (flop b)
$(a t.a, b [?:(=(4 i.a) 3 i.a) b])
```

4 Attacks in Urbit

Nock and Hoon are homoiconic: code and data share the same representation. This facilitates code distribution over Urbit's Ames network, wherein code is transmitted as a raw jammed noun (nominally in Hoon) and then built using local mark conversion, parsing, and compiler routines. The local ship controls its own build process rather than accepting unknown Nock nouns, permitting the source code to be inspected (often in abbreviated form by comparing "desk hashes" or hashes of the contents of a distributed branch) prior to installation.[6] This suggests at least three different programmatic attack vectors for an Urbit ship:

1. Malicious source.
2. False signatures.
3. Compromised compiler.

Each of these could be perpetrated by the last two: the idea of a compromised compiler trumps all bets for a program, and thus prompted Thompson's initial reflections.

[6] In practice, we observe that the relatively high trust of the Urbit community has led to the hash being used as a version identifier rather than a cursory security check.

4.1 Malicious source

The most straightforward way to compromise a system is to simply send code that compromises a system in a particular way and induce the target to execute that code. As Urbit code distribution is permissionless, the practical barriers to this are social and habitual: most users prefer to treat the Urbit network as a high-trust society, and most users do not store high-value data (such as wallet keys) on their Urbit ship. However, by representing code as valid, an attacker can execute at least on zero-day exploit against a target.

Furthermore, userspace code distribution does not have permission to modify the `%base` desk on which the kernel resides. Permission elevation would not be necessary for an attack, however; malicious code could sidestep the per-desk sandbox mechanism by simply directly issuing cards to Clay to modify `%base`'s `/sys` files.[7] Such would be easy to identify in program source, but would require some diligence on the part of users to hedge against.

Using a social trust mechanism to convince a user to utilize a bad `|ota` source is by far the simplest mechanism, however.

4.2 Signature falsification

A noun hash is a cryptographic hash of a noun. Specifically, the SHA-256 `++shax` of an atom is typically used, or for cell structures the noun is first `++jam`med into an atom before a SHA-128 `++shaf` is applied. Noun hashes are used extensively in Arvo and the vanes to identify values in caches and to compare nouns for equality.

In principle, signature falsification could lead to invalid cache retrievals, e.g. invoking a malicious gate instead of the intended one. In practice, userspace code does not have any direct access to the Arvo caches, meaning that while signatures

[7] At the time of writing, `~hastuc-dibtux` pointed out that the `%hood` agent itself can be used to directly rewire the source. More aggressive countermeasures and a permissioning system will be necessary as Urbit hardens its security model.

could be falsified, this would require a kernel-level or runtime-level compromise of the system.

Ames packet signature falsification Over the wire, there are two significant kinds of risky false signatures: signed Ames packet or faked desks. An Ames communication takes place as a message is decomposed into 1 kB or smaller packets, each of which is sequentially numbered and signed. All packets are signed using SHA-256 encryption.[8] On the other end, the packets are received in a queue before being reassembled into the original message. We consider three different signature attacks:

1. Falsifying a packet signature. To simply falsify a signature, one would have to falsify the private key of the sender, which is a significant cryptographic challenge.
2. Spoofing a packet signature. A locally compromised runtime could inject a spurious packet matching some hypothetical external ship. This attack is of commensurate difficulty to a simply falsified signature.
3. Permitting a false signature. A compromised Ames vane could maliciously permit a message with a false signature, perhaps activated by the signature itself. This scenario seems the most tractable for a hacker to exploit.

In practice, packet reassembly takes place in the runtime, which introduces another attack surface for these Ames packet signature falsification attacks.

Desk (Clay) signature falsification A desk hash, used as an informal check,[9] is a signature from the contents of the desk.

We may dismiss the problem of directly producing a malicious source desk that hashes to the same value as a highly nontrivial expenditure of compute resources. In addition, should such an attack come into currency, the hash algorithm could

[8] "Every packet sent between ships is encrypted except for self-signed attestation packets from 128-bit comets" ("Ames Overview").

[9] `(shax (jam +<))`, where the referent is a list of commits.

easily be traded for a more complicated and secure one. For a zero-day exploit utilizing the distribution of a compromised desk, prior art suggests that on the order of $2^{63.1}$ SHA operations would be required to construct a collision (Stevens et al., 2017). (Successfully execution would then require that the malicious desk actually contains malicious code, which further complicates the problem space.)

As with the Ames packet signature falsification, it is more likely for malicious desk distribution to succeed via a special-cased desk than by a hash collission. This has an inception problem, of course, since the `%base` desk must be likewise modified (or the threshold attack described below used).

4.3 Compromised compiler

As with pattern-matching examples in the exposition, simple or complex pattern-matching to inject and perpetuate malicious code is a possibility given a Turing-complete language and execution platform. Hoon is of course no exception, so a compromised compiler may alter any outer cores in the current system state and any future nouns produced by the suspected compiler.

Source is of course amenable to inspection, but the trusting trust attack is complete once the compiler is compromised. Thus a compromised compiler is a local problem more than a network-wide risk. The most sensible detection mechanism is to continue rigorous use of hashes to identify nouns and their source uniquely.

5 Attacks in the Runtime

The runtime operates as the computation substrate for Arvo and could execute arbitrary malicious code if compromised. Nock is an interpreted language which is evaluated on some particular virtual machine layer. As of this writing, there are many Nock interpreters **Nock2023** but only two fully instrumented Nock runtimes, Vere and Sword (née Ares). Vere is written in C and utilizes a few third-party libraries statically linked into the binary, such as `libuv`. Sword is composed in Rust and similarly statically links its modules. Per the concerns raised by Thompson about a program produced by a compromised compiler, the runtime could in principle be modified to run malicious code either independently of the Arvo instance or in response to side effects induced by the Arvo instance.

The colloquial wisdom among the Urbit developer community is that if one loses administrative control of one's box (the machine executing the Urbit process), one is already pwned.[10] This is saliently illustrated by the Trusting Trust compiler issue, but of course there are less subtle and more

By inducing a user to install a compromised Urbit runtime, an attacker can achieve full control over a user's ship and its computation, perhaps without the user being aware. Three major components of the runtime could be used to compromise an Arvo instance:

1. Nock evaluation.
2. Cache poisoning.
3. Hint pollution.

Nock evaluation. Nock evaluation in the runtime takes place as some form of bytecode evaluation, rather than evaluating the pure Nock formula. (Subject-knowledge analysis (SKA) further complicates the relationship of on-chip evaluation to the source Nock.) Nock evaluation could be matched for specific patterns and malicious code executed, or simply

Cache poisoning. Like Arvo, the runtime uses caches and the event log extensively to hasten referentially transparent evaluation. We consider two examples, of many possibilities, in which cache poisoning could be used to compromise an Arvo instance.

The first is to compromise the jet state in event playback. (This is a form of "hint pollution.") Jets formally do "nothing" in

[10] A related problem is that of the security of Azimuth as a public key infrastructure, but we do not address this line of enquiry in this treatment.

Arvo and are used as a way of creating impure side effects for the Arvo operating function. A compromised jet hint would be a Turing-complete compromise, and incorrect matching of an event playback could result in any possible side effect which the runtime can achieve on the host OS. The user may not be aware of any malicious code execution. Furthermore, due to the way that Vere's jet dashboard matches code, a registered noun can have its associated jet hint removed in the Hoon source but still be invoked due to noun matching; this is a leaky convention that could lead to jet pollution.

The second is to simply poison known caches such as the remote scry cache. Malicious code could thus be emitted to a subscriber *without the host even being aware of the poisoned cache value.* Another upcoming cache susceptible to poisoning is the fastboot cache used to supply compiled nouns in the process of bootstrapping a new ship. Finally, as explicit caching in Arvo is gradually replace with persistent memoization in the runtime, the runtime and its distribution mechanism will need to be provisioned with security assurances for users.

Hint pollution. There is a particular threshold attack which we must consider in detail: the injection of a Nock Eleven flag (with no Hoon side effects, rather evocative of C `#pragma`s but not the same) that nevertheless triggers malicious code execution or exfiltration of sensitive information from the running Arvo instance. A hint is a noun that is used to guide the evaluation of a Nock formula. The compromise can be in the runtime but the trigger in this case comes from the Arvo side.

The most salient attack vector on an Urbit ship, of course, is to compromise the cryptography: emit the Azimuth private key or enough information for an observer to deduce the key. Several of the foregoing techniques, jointly or separately, could be used to achieve this.

In general, of course, the solution is to be paranoid about one's runtime and one's Arvo; if either is discovered to be compromised, it is high time to replace the one and breach the other.

6 Conclusion

As Urbit moves towards automated update processes, such as acquiring and booting into new runtimes directly, secure control over the stack—including the build—will assume new prominence. We hope these reflections will spur more careful consideration of the dangers inherent in running and securing not only Urbit but in fact any computing stack. Fortunately, even at the current time aggressive checking of Urbit (including building one's own runtime binary and checking various hashes proactively) can and should make the likelihood of a trusting trust attack as low as possible, since each check in the process makes it increasingly more difficult to propagate malicious code from previous steps correctly. Such affordances should be propagated into the end-user experience as soon as feasible.

References

Stevens, Marc et al. (2017). *The First Collision for Full SHA-1*. Tech. rep. Google Research. URL: https://web.archive.org/web/20180515222208/http://shattered.io/static/shattered.pdf (visited on ~2024.8.12).

Thompson, Ken (1984). "Reflections on Trusting Trust." In: *Communications of the ACM* 27.8, pp. 761–763. DOI: 10.1145/358198.358210.