Security Considerations in a Forth Derived Language

Ron Aaron
Aaron High-Tech, Ltd.
HaSheminit 34, Ma'ale Adummim, Israel
t: +972 52 652 5543
e: ron@aaron-tech.com
w: https://8th-dev.com/

Abstract

Of major concern in modern application development is how to increase the robustness of program code while maintaining the programmers' productivity. Time-to-market directly affects ROI, but so do security problems arising from coding practices and tools. Our "8th" language, a Forth derivative, was conceived in part to address these issues.

Introduction

In seeking a cross-platform and secure programming language for a "secure application" which was to run on both mobile and desktop platforms, I looked for a language which was:

- easy to use
- truly "write-once, run-anywhere" (mobile and desktop alike)
- resistant to hacking
- proof against the most common security problems

Because I didn't find an appropriate language or set of tools, despite my best efforts to do so, I settled on creating a new language. Borrowing ideas from my former Reva Forth, which was also a Forth-derived language, I set out to address the above issues in the new language, "8th".

"Ordinary" security issues

Memory access:

In standard Forths, it is generally possible to access any memory in the system by means of `@` and `!`. Similarly, in C and C++, it is possible to assign an arbitrary value to a char * to do the same. While it is often useful to have such access, it is an open door for hackers to subvert a program. Therefore 8th does not permit arbitrary memory accesses. Instead, it uses strong typing along with specific memory-accessing types (strings and buffers, among others) to safely encapsulate memory accesses.

Memory allocation:

A common issue in C, C++, Forth, and others is memory allocation. There are several related issues: a) allocating an incorrect amount of memory, b) neglecting to properly initialize that memory, c) neglecting to allocate memory at all, d) forgetting to de-allocate the memory when done, and e) de-allocating the memory more than once. Problems (a), (b) and (c), in particular, can also be security issues. The 8th solution is simply to not permit the programmer to allocate or deallocate memory (much like Java *et. al.*). The data-types native to 8th automatically manage any memory they require. They are also allocated from "pools" of that type, and they are reference-counted so that de-allocation of memory (and other resources) occurs when the item is no longer in use. This sort of "garbage-collection" amortizes the cost of the cleanup over the course of the application's run-time, similar to C or C++ (or Forth for that matter), and unlike Java whose GC can cause the application to pause for a significant amount of time at unpredictable intervals.

Stack smashing:

Possibly the most common exploit used by hackers is "stack smashing", or overflowing an item held on the stack and thus overwriting adjacent memory. This can be used to simply corrupt a value (and achieve some alternate code path), or it may be used to generate an alternate code address to return to, or even CPU opcodes to be executed.

Despite decades of battle against this particular foe, C and C++ remain quite vulnerable to it because the languages themselves make it easy to succumb to it. 8th makes it impossible to smash the stack, since data-items control access to their internal data, and ensure the access is within bounds, or dynamically grow as necessary. In no case is direct access to the CPU stack provided by any 8th words, unlike the standard Forth >r and r> which, often, are implemented as giving direct access to the CPU stack. In 8th, those words only access an auxiliary stack, so it is impossible to subvert the CPU stack.

Integer overflow:

While perhaps not as well-known, integer overflow continues to be an ongoing security issue in C and C++ in particular (as well as in some Forths). The plethora of integer types and the necessity to always be aware of those types, as well as the overuse and abuse of "casting" can lead to unexpected and extremely difficult-to-debug security issues. 8th handles this by automatically promoting an integer to a "big-integer" or "float" to a "big-float" as necessary. Adding 1 to 9223372036854775807 simply results in 9223372036854775808, and the underlying data type appears simply as a "number" to the user. This means that the programmer can be concerned solely with the correctness of the algorithm he or she is implementing, and needn't be worried about "gotchas" in the maths department.

Pointers:

Forth users are used to having generic "cells" on the stack, which may be treated as numbers or addresses (arbitrary memory access). C/C++ users rely heavily on pointers. The danger is that a pointer represents an open door into your data structures and perhaps your code as well. If an adversary can get access to a pointer to an internal structure, she or he can almost certainly find something "interesting" to do with it. Something which you will not approve of, most likely.

8th does not provide any pointers at all. This was hinted at in the “memory access” section. But it goes further, in that the data items returned by 8th to the user are opaque and it is not possible via plain 8th to access their internals. However, since the FFI (foreign function interface) must be able to pass pointers to external functions, it is possible for an external library to gain access via the FFI to internal structures. 8th attempts to reduce the footprint of this vulnerability by not passing any actual internal structures to external programs; however, a dedicated hacker might be able to find an opening. Therefore one must take extra steps to ensure external libraries are secured).

User input:

Another avenue for hackers to subvert an application is via uncontrolled (or more often, specifically controlled) user-input. In common with Forth, 8th allows users to enter arbitrary information via the REPL. The input in 8th is captured in strings, which are a self-adjusting data-type which can handle any amount of data (within system memory limits). In general, the REPL is not considered a vector for subverting the application. However, the application programmer can use **eval** to permit interpretration of arbitrary input text, or can even invoke the REPL from within an application.

To help the programmer avoid disaster, 8th makes it quite easy to limit the vocabularies (called “namespaces” in 8th) which are available to the REPL. Standard Forth has a similar facility. Thus one may control which words the user-interpreted input may access.

But unlike standard Forth, 8th uses JSON as its data-description language, primarily because it is so widely used and is easy to read. Further, 8th has “enhancements” to JSON which are intended as a convenience to the programmer. So for example, one may embed 8th code to be interpreted along with the JSON, in order to calculate some value at load-time. These enhancements are a potential avenue for hackers, and so a special word **json>** exists which interprets only standard JSON (and throws an exception if there is invalid JSON in the string).

In the usual case of user-input hacking, the hacker attempts to overflow a buffer in order to perform a stack or heap-crash. Because the strings used by 8th are self-adjusting, this tactic cannot succeed.

An additional protection 8th provides is “auto-wipe” buffers and strings. When one of those data types has **set-wipe** invoked on it, it will have its allocated memory wiped (zeroed out) before being deallocated. This is important in the case of a buffer holding decrypted information or an encryption key. The encryption words set the wipe flag automatically to ensure private or sensitive data are not leaked.

“Extraordinary” security issues

There are two more specific issues which 8th attempts to deal with: subversion via required libraries and validation of the application's code.

In any system where external libraries may be loaded at run-time (e.g. all the platforms 8th runs on), it is possible for an adversary to replace a “good” library with a “bad” one. Windows has been a non-stop target of such exploits, but neither macOS nor Linux are immune. 8th reduces the attack surface by fully incorporating most of the libraries it requires into the runtime binary. This has the obvious disadvantages of increasing the size of that binary, as well as making it impossible to update the version of a library used in a particular 8th binary. However, it was felt that the advantages of significantly reducing the attack surface for this kind of hack outweighed those disadvantages. In particular, because 8th undergoes very regular updates, the incorporated libraries are also regularly updated as necessary.

The “Professional” edition of 8th also incorporates validation of the application as well as a measure of protection against decompiling or exposure of the programmers intellectual property. It does this by encrypting and signing the “deployment package” (DP) of code and resources.

In practice, the DP is a zip-file containing all the code and resources for a particular application. The Professional edition allows programmers to encrypt the DP (with AES-256-GCM) and sign the application with their specific PK keys (generated by the 8th system using Ed25519, not GPG etc.). Then when the 8th runtime unpacks the DP, it will fail if any part of it has been tampered with, providing a strong measure of confidence that the DP is legitimate. The DP's encryption makes it difficult (though not impossible) for an adversary to decrypt it and expose the intellectual property contained therein.

Conclusion

There is no rest for the weary.

We are continually seeking to improve the hack-resistance of 8th and improve its stability. At present, the FFI represents the single biggest entry point for hackers, and we are researching methods of hardening it.

Likewise, we would like to find a better method of encrypting the DP, to make it still more difficult for an attacker to decrypt. The weakness there is that the keys must be known at runtime, and so must be stored in a manner the 8th runtime can access on arbitrary hardware. We've not found any particularly robust solutions to that problem, so we are careful to not make any extreme claims for the encryption.

As new exploits are found in libraries we use or in other code, we work to find ways to offset them.

Finally, we are researching methods of confounding timing and other attacks against the encryption libraries we use (TomCrypt).

If you have comments, questions, or criticisms, please feel free to address them to me at the email listed at the start of this document.