|  |  |
| --- | --- |
| TEA PRIMITIVE | SEMANTICS |
| E: | |  |  | | --- | --- | | NAME | Evaluate | | PURPOSE | Process memory as though it were a TEA program | | SYNTAX  & SEMANTICS | |  | | --- | | e: | | [*Context Unaware Evaluation*] Process the contents of AI as an external TEA program, with its initial AI as the EMPTY STRING and set the final IO from that program as the IO for the current instruction | | e:STR | | [*Context Aware Evaluation*] Process the contents of STR as an external TEA program, but with current AI as its initial AI and set the final IO from that program as the IO for the current instruction | | e!: | | [*Extended* *Context Evaluation*] Parse the contents of AI as a TEA program, and inject the found TEA instructions in it inline into the calling main TEA program---essentially, modifies the main program by injecting new code into it, replacing the original E: instruction in the calling program with the new program instructions, then continues normal processing in the main/calling program, starting from the first instruction in the newly injected instructions if any – the EMPTY STRING becomes input to that first instruction, otherwise (in case no TEA instructions were found) continues to the next instruction in the calling, unmodified program, with the AI set to the EMPTY STRING. | | e!:STR | | [*Extended* *Context Aware Evaluation*] Same as E!:, but passes current AI to the first instruction in the extension/injected TEA program, otherwise (in case no TEA instructions were found) passes it on to the next instruction in the main/calling TEA program unmodified. | | e\*:  e\*!: | | INERT | | e\*:vNAME | | Same as E:STR, but processing string stored in vault named vNAME | | e\*!:vNAME | | Same as E!:STR, but processing string stored in vault named vNAME | |  | | NOTES | One could appreciate **E:** by the following illustrative examples:  i!:{i!:AAA|d:^A|r:$:W} | e:  i!:{i!:AAA|d:^A|r:$:W} | e!:  # or...  i!:{BC CB BA AB} | e:"i!:AAA|d:^A|r:$:W" | # note the “” emphasis  i!:{BC CB BA AB} | e:{i!:AAA|d:^A|r:$:W} | # explicit context unaware program in STR  i!:{BC CB BA AB} | e!:{i!:AAA|d:^A|r:$:W} | # explicit context aware injected program  i!:{BC CB BA AB} | v:vPROG:{i!:AAA|d:^A|r:$:W} | e\*:vPROG # context-unaware vault program  i!:{BC CB BA AB} | v:vPROG:{i!:AAA|d:^A|r:$:W} | e\*!:vPROG # injection with vault program  i!:{BC CB BA AB} | v:vPROG:"i!:AAA|d:^A|r:$:W" | e\*!:vPROG # note the “” emphasis  # all variants above should return (="AAW")  The above eight TEA programs are actually equivalent though different and all approaching the same problem – essentially, executing external/injected TEA code from within a main/host TEA program. However, to truly appreciate the power of E:, let us consider the following non-trivial program:  i!:ABC  l:lHEW  h:  e!:{  v:  v:vE:XYZ  |v!:  |v:vOL  |g\*:-< >-:vE:vOL  |v:vMIX  |l:lGEN  |p!:5  |x\*:vMIX  |f:[ai]:lMASK  |l:lSALT  |s:1\_0\_1  |f:[a1]:lGEN  }  l:lMASK  g!:\*\*  It not only demonstrates all the tricky aspects of a TEA program, however, it is a great example for how to create self-modifying TEA programs. Essentially, all the TEA code inside the e!:{…} block ends up not being processed as an external TEA program, but gets injected into the main program, and uses its AI and existing label blocks! This program will for example return a string starting with the injected sub-string “XYZ” such as “XYZ\*\*5uaikq” if the injected program reached a state where the AI contains the sub-string “ai”, otherwise will return a string starting with “A++B++C” such as “A++B++CXYZ-< >-5hlca1\_0\_1x” in case the program reached a state where the AI contains the substring “a1”. A full appreciation of how this program works is shown in a runtime DEBUG dump of this program in the official CLI TTTT tests [3]. |  | |