WebAssembly Security

Analysis of Google Keep WebAssembly module

Last month, i was at REcon Montreal to give my training about WebAssembly Security and after some discussion people always ask me this question:

Is WebAssembly already used in the wild?

The answer is of course YES and some WebAssembly modules are potentially running right now in your browser if you are using Google web services. Recently, Google was using WebAssembly for the beta version of Google Earth but also in production for services like Google Keep.

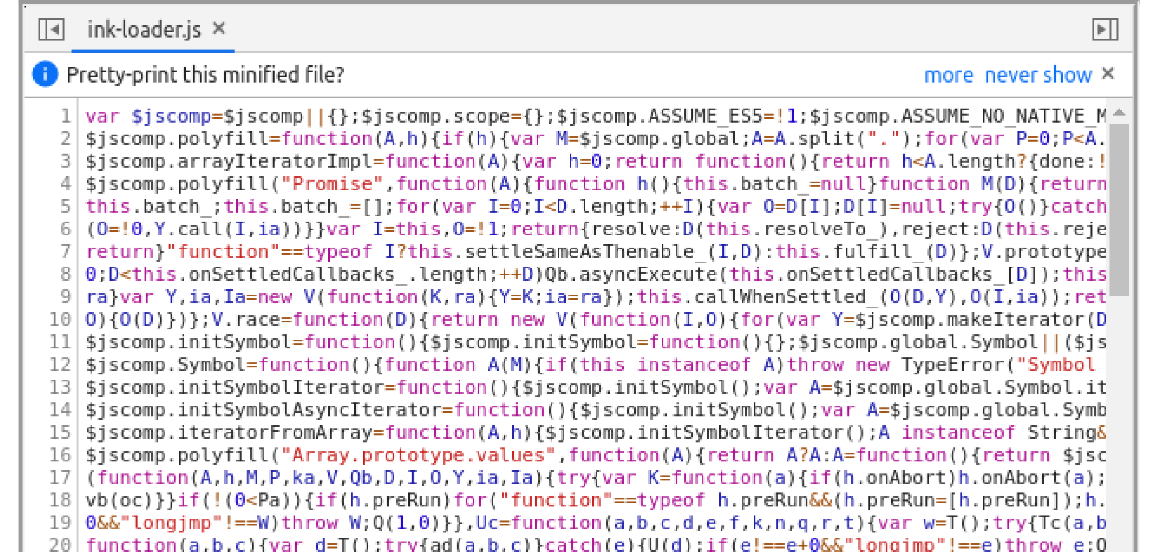
In this blogpost, we gonna reverse partially the WebAssembly module loaded by Google Keep, determine its purpose and extract a maximum of information for future complete analysis. Let's Go!

Google Keep Wasm Module & JS File Extraction

Usually, in order to run a WebAssembly module in your web page, you will fetch a wasm file and instantiate the module using dedicated JavaScript API. Once it's done, you will be able to call the module exported functions directly from JavaScript.

Regarding the Google Keep web app, the WebAssembly module "**ink.wasm**" is fetch (image below – left) and instantiated by the minified JavaScript file **ink-loader.js.** (image below – right)

Based on JS functions names, this JavaScript file seems to has been generated automatically by emscripten.



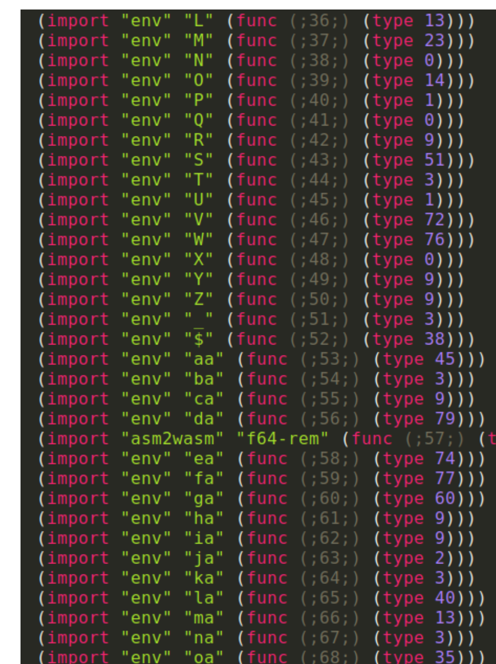
WebAssembly Module Reversing

One of the first step when reversing a WebAssembly module is to convert the wasm binary (.wasm) to his text format (.wat/.wast) representation. wasm2wat is the perfect tool for this job.

wasm2wat ink.wasm -o ink.wat

The output file (**ink.wat**) is a text file with around 1.5 Millions of lines.

Based on minified imported & exported function names (image – right), we can confirm that the module has been compiled by emscripten and the optimization flag (-O3)



Extract Build Information

This module contains a Data section and the content of this section will be used to initialized the linear memory i.e. an ArrayBuffer shared between the module and the loader script (**ink-loader.js**).

```
(data (;0;) (i32.const 1024) "254950189")
(data (;1;) (i32.const 1056) "-1")
(data (;2;) (i32.const 1088) "build-secure-info:source-uri")
(data (;3;) (i32.const 1600) "1")
(data (;4;) (i32.const 1632) "emscripten-wasm")
(data (;5;) (i32.const 2144) "Jun 25 2019 06:29:13")
(data (;6;) (i32.const 2176) "//depot/google3")
(data (;7;) (i32.const 2688) "63c1dd42-a3ab-4e17-8014-b4978811fc13")
(data (;8;) (i32.const 3200) "docs-release@votl8.prod.google.com:/google/src/files/254950189/depot/google3")
(data (;9;) (i32.const 3712) "memento_2019.26-Tue-0616_RC00")
(data (;10;) (i32.const 4224) "//third_party/sketchology/public/js/wasm:ink.js")
(data (;11;) (i32.const 4736) "1561469353")
(data (;12;) (i32.const 4768) "Blaze, release blaze-2019.06.17-2 (mainline @253503028)")
(data (;13;) (i32.const 5280) "blaze-out/wasm-opt-8ef3d7d56bdc46b9c241441ae1d9670d/bin/third_party/sketchology/public/js/wasm/ink.js")
(data (;14;) (i32.const 5792) "1")
(data (;15;) (i32.const 5824) "plain")
(data (;16;) (i32.const 5856) "254950189")
(data (;17;) (i32.const 5888) "none")
(data (;18;) (i32.const 5920) "docs-release")
(data (;19;) (i32.const 6432) "votl8.prod.google.com")
(data (;20;) (i32.const 6944) "/google/src/files/254950189/depot/google3")
(data (;21;) (i32.const 7456) "Built on Jun 25 2019 06:29:13 (1561469353)")
(data (;22;) (i32.const 7968) "changelist 254950189 with baseline 254950189 in a mint client based on //depot/google3")
```

At the beginning of this module Data section, we get a lot of details about how the module has been built:

- WebAssembly Toolchain: **emscripten-wasm**
- Building date: **Jun 25 2019 06:29:13**
- Google prod server: **votl8.prod.google.com**
- Project path: **third_party/sketchology/public/js/wasm**
- Google building software (Bazel): **Blaze, release blaze-2019.06.17-2**

At this point, I first tried to retrieve the source code of the WebAssembly module by searching the project path on the web. I found the repository of Chromium 66 (66.0.3359.158 ~1 year old) but without C/C++ source code inside. On the master branch, there is no reference of sketchology anymore but we get information about what is Ink. Finally, the github repository (**https://github.com/google/ink**) return a 404 error.

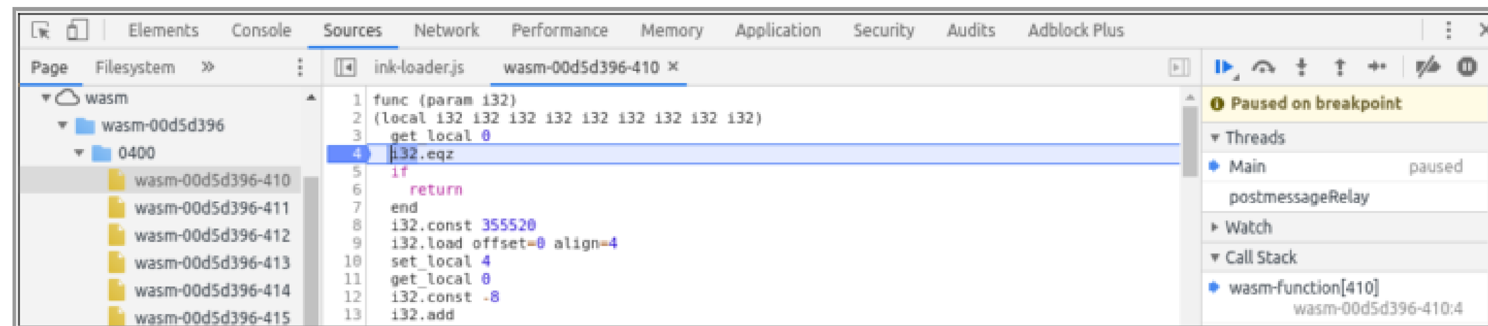
What is Sketchology and Ink?

After some research, Sketchology refers to an IOS application called “Sketchology Review”. This application isn’t available anymore, the twitter account is inactive and the official website (sketchologyapp.com) is down but you can find a copy using the WayBack Machine. On the LinkedIn’s profile of the creator, we can find that *“Sketchology is the first vector drawing app with realtime natural media brush effects like blur or watercolor.”*

On the other hand, *“Ink is a software library enabling Google applications to let their users express themselves using freehand drawing and handwriting”*. This library is also used in Google Canvas released end of 2018 (source).

So, it seems that Ink is the evolution/successor of Sketchology and Google Keep use this module ink.wasm when the user want to draw a note (image on top).

To verify our hypothesis, you can debug the WebAssembly module and set breakpoints using the Developer console. In the image below, my breakpoint was triggered when i tried to create a new drawing note.



Reversing Protobuf Encoded Blobs

```
- 0002e80: 0a2a 7468 6972 645f 7061 7274 792f 736b  .*third_party/sk
- 0002e90: 6574 6368 6f6c 6f67 792f 7072 6f74 6f2f  etchology/proto/
- 0002ea0: 6269 746d 6170 2e70 726f 746f 1209 696e  bitmap.proto..in
- 0002eb0: 6b2e 7072 6f74 6f22 470a 0642 6974 6d61  k.proto"G..Bitma
- 0002ec0: 7012 0d0a 0577 6964 7468 1801 2001 280d  p....width.. .(.
- 0002ed0: 120e 0a06 6865 6967 6874 1802 2001 280d  ....height.. .(.
- 0002ee0: 121e 0a16 636f 6d70 7265 7373 6564 5f62  ....compressed_b
- 0002ef0: 6974 6d61 705f 626c 6f62 1803 2001 280c  itmap_blob.. .(.
```

Still inside the module data section, you will find multiple chunk of Google Protobuf encoded blobs (image on the top).

“Protocol buffers are Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.” – source

Those chunk of bytes can be reversed/deserialized using tools such as protobuf-inspector (image at the bottom). Source code of the more generic protocolbuffer file can be found directly on the github repository of the protobuf project (like descriptor.proto)

This kind of information is particularly useful if your are doing pentesting/vulnerability research on the server-side web API.

```
\  ~/Documents/reverse_tool/protobuf-inspector/main.py <  protobuf/bitmap.chunk
root:
    1 <chunk> = "third_party/sketchology/proto/bitmap.proto"
    2 <chunk> = message:
        13 <64bit> = 0x6F746F72702E6B6E / 8031166572807482222 / 7.7456628e+228
    4 <chunk> = message:
        1 <chunk> = "Bitmap"
        2 <chunk> = message:
            1 <chunk> = "width"
            3 <varint> = 1
            4 <varint> = 1
            5 <varint> = 13
        2 <chunk> = message:
            1 <chunk> = "height"
            3 <varint> = 2
            4 <varint> = 1
            5 <varint> = 13
        2 <chunk> = message:
            1 <chunk> = "compressed_bitmap_blob"
            3 <varint> = 3
            4 <varint> = 1
            5 <varint> = 12
```

Extract WebGL Vertex Shader Structure

Another part of the data section contains complete piece of codes (bottom image) with variables and main functions. This code is a WebGL “Vertex shader structure” and it will be loaded by WebGL building shader functions at runtime.

```
float make_cyclic(float value, float start, float period) {
  return mod(value - start, period);
}

void main(void) {
  // Components for \22gravity\22 and initial velocity. (world coords)
  float g = -450.0;  // Determined empirically.
  float vx = velocity.x;
  float vy = velocity.y;

  // Make the color time cyclic.
  float modTime = make_cyclic(time, sourceColorTimings.x,
                              sourceColorTimings.y - sourceColorTimings.x);
  float colorLerpAmount =
      clamp(modTime / (sourceColorTimings.y - sourceColorTimings.x), 0.0, 1.0);
  destcolor = mix(sourceColorFrom, sourceColorTo, colorLerpAmount);

  // Make the position time cyclic.
  float t = make_cyclic(time, positionTimings.x,
                        positionTimings.y - positionTimings.x);

  // Perform transforms in world coords.
  vec3 world_position = object * vec3(position.xy, 1);
  world_position.x = vx * t + world_position.x;
  world_position.y = g * t * t + vy * t + world_position.y;

  textureCoordsOut = textureCoords;
  vec3 homogeneous = view * world_position;
  gl_Position = vec4(homogeneous.xy / homogeneous.z, 0, 1);
}
```

If you want to learn more about WebGL and Vertex shader, take a look at those links:

- Compiling a C++ OpenGL Project for OS X and WebAssembly
- WebGL Fundamentals
- The Book of Shaders
- Canvas filled three ways: JS, WebAssembly and WebGL
- What Do You Mean by “Shaders”? How to Create Them with HTML5 and WebGL

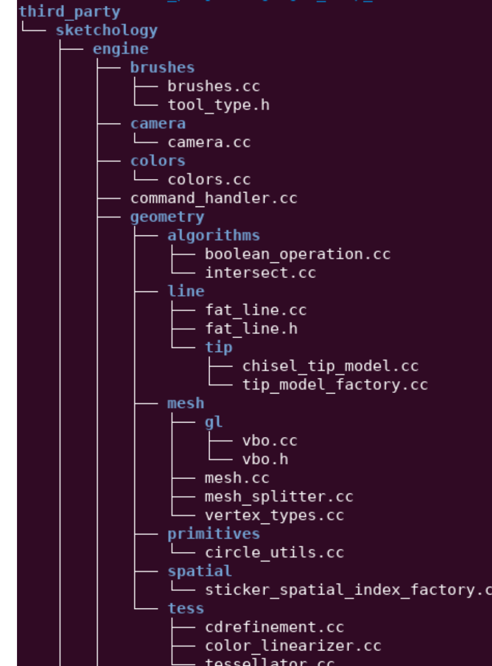
Absolute path, Error messages, Mangling & Constant names

Finally, we reach the last part of this module data section that is for me the most interesting one. Inside you will find more than 5 thousands strings like:

- Absolute project files path (**“third_party/sketchology/engine/public/sengine.cc”**)
- Error messages (**“Could not add image data, no URI specified.”**)
- Mangling functions name (**“N3ink26ElementAnimationControllerE”**)

- Constant names (**“GL_GEOMETRY_SHADER”**)

Just with those strings, we can reconstruct the project tree (image on the left) and associate the corresponding error messages, mangling names and constants for each file.



Going Deeper & Conclusion

```
tee_local 0
i32.const 314317 ;;Cannot allocate buffer larger than kint32max for
i32.const 49
call 412
drop
get_local 0
i32.const 314367 ;; StringOutputStream
i32.const 19
call 412
```

If you (really) want to reverse completely this module, you will need first to match the previous information (WebGL, debug strings, …) with memory accesses/offsets (image on the top).

Then, you can determine the functions prototype (mangling names + arguments) and associate each WebAssembly functions with C++ source files. Finally, you can try to decompile your new labeled module into C code using tool like wasm2c.

Nevertheless in this blogpost, we have at the end:

- **Extract a WebAssembly module and related JS file.**
- **Convert a module to the text format representation.**
- **Found build information**
- **Determine the origin and the purpose of the module.**
- **Reverse Google Protobuf encoded blobs.**
- **Extract WebGL shader source code.**
- **Reconstruct the project tree**

- **Find debug strings to reverse completely the module (with more time)**

All the files (wasm, js) and extracted information are available in this github repository.

If you want to learn about WebAssembly security from module reversing to WebAssembly VM vulnerability research, you should consider taking one of our trainings. We also offer on-site trainings for companies, starting at just 5 participants.

Patrick Ventuzelo / @Pat_Ventuzelo

© 2019 - Patrick Ventuzelo | Contact | Trainings