Reverse Engineering MicroPython Frozen Modules

Data Structures, Reconstruction, and Reading Bytecode

Dr. Wesley McGrew
Senior Cyber Fellow

w.mcgrew@martinfed.com

martinfed

Agenda

- Introduction to MicroPython and frozen modules
- MicroPython firmware on Raspberry Pi Pico
  - Removing metadata
- Building Hello World
  - Compiling it into .mpy bytecode
  - Freezing it into the code
- Hands-on example: Reconstructing Hello World .mpy from stripped binary
- Analysis
- Conclusions

Introduction

- **MicroPython -** An implementation of Python 3 for microcontrollers
  - Subset of Python standard libraries
  - Some libraries that are nice for working with microcontroller hardware
  - REPL
  - Used in some commercial/scientific/industrial products and projects
  - **#BADGELIFE**
  - Pretty fantastic stuff for quick prototyping and tinkering around
- Code can be
  - Plain text (.py) (a small filesystem is provided that you can interact with)
  - Pre-compiled into bytecode (.mpy)
  - ***“Frozen” into the MicroPython firmware at build-time***

Frozen Modules

- This feature is here for efficiency and convenience
- ...but may be mis-used for obfuscating/hiding code and secrets
- Some quotes

Security through Obscurity

Not really secure, but can lower the risk to the causal thief

- You can frustrate attempts to gain access to your secure keys by using special functions and using pre-encoded usernames and passwords - the device can decode these, without the attacker easily understanding them
- We can also compile MicroPython code into MPY files, which is a binary format. If you open an MPY file, you will see a mixture of byte-code as well as plain text from the code
- We need to hash these strings to make them indistinguishable from the byte code
- Its not ***really*** secure, but its better than no security - the tenacious hacker can already gain access to your wifi username and password through tools like Sharkwire
- Its about risk management

How can you make copying the code more difficult?

It depends on what code you want to make difficult to copy and whether you need to prevent copying or just want to prevent reverse engineering of what that code is.

You can tokenise MicroPython code into a .mpy file which will make that more difficult to reverse engineer when shared - But only until someone cares to work out how to reverse engineer it,

We care ;)

Those '.mpy' files are different to '.pyc' files and MicroPython uses different bytecode to CPython so disassemblers which exist for that don't work. It is presumably feasible to create a disassembler for MicroPython but I am not sure anyone has.

How can I hide my codes?

OpenMV | Hi, we plan to roll out the ability for you to use frozen MicroPython code soon. This will let you compile the code into a byte image that we'll be able to read.

Note that anyone can easily reverse engineer the byte image to get some ugly source code… But, in general folks won't.

Frozen Modules

There are a few options:
- Use mpy-cross to make a .mpy file out of your .py files and put them on the filesystem instead. (They'll be bytecode instead, so someone can't just open them up and read your code text)
- "Freeze" your Python code into the firmware image. (This is a good idea anyway as it means your program will use less RAM)

You could put the configuration in a Python module file and add that to the frozen modules. Then someone trying to get at it would need to read out the RAM or the flash memory of your board, which makes this a bit more inconvenient but will not detract determined attackers. Also it makes it difficult to change the password on-the-fly.

@iotmaker A simple method would be to make main.py very primitive and put the code in some other script, let's call it myapp.py, which will be embedded as frozen bytecode. main.py would then simply consist of:
import myapp
Frozen bytecode would require you to build your own image, which is not that hard. And frozen bytecode is like compiled code. Text and constants are visible in clear, and it is possible to create wome kind of decoder. But it is some barrier.
Putting main.py literally is not reccomended.

The general vibe is: It’s not perfect, but there’s nothing “out there” about reversing it, so it’s a layer of protection

Getting a firmware image

- Out of scope for this talk, and potentially tricky for microcontrollers that protect it
- If you're following along at home with a Pico, it's easy:

```
wes@thinkpad:~/pico/mp_build_stock$ sudo /home/wes/bin/picotool info -a
Program Information
 name:              MicroPython
 version:           v1.24.0-preview.34.g5903ee561
 features:          thread support
                    USB REPL
 frozen modules:    neopixel, dht, ds18x20, onewire, uasyncio, asyncio/stream, asyncio/lock, asyncio/funcs,
                    asyncio/event, asyncio/core, asyncio, _boot_fat, _boot, rp2
 binary start:      0x10000000
 binary end:        0x10050bd4
 embedded drive:    0x100a0000-0x10200000 (1408K): MicroPython
```

```
wes@thinkpad:~/pico/mp_build_stock$ sudo /home/wes/bin/picotool save -r 0x10000000 0x10050bd4 -f extracted_firmware.bin
Saving file: [==============================]  100%
Wrote 330708 bytes to extracted_firmware.bin
```

Picotool “Binary Information”

- Picotool *README*: “Binary information is machine locatable and generally machine consumable. I say generally because anyone can include any information, and we can tell it from ours, but it is up to them whether they make their data self describing.”
- MicroPython *ports/rp2/mpconfigport.h*

```
#define BINARY_INFO_TAG_MICROPYTHON BINARY_INFO_MAKE_TAG('M', 'P')
#define BINARY_INFO_ID_MP_FROZEN 0x4a99d719
#define MICROPY_FROZEN_LIST_ITEM(name, file) bi_decl(bi_string(BINARY_INFO_TAG_MICROPYTHON, BINARY_INFO_ID_
```

Picotool “Binary Information”

- MicroPython *tools/mpy-tool.py*

```
# If a port defines MICROPY_FROZEN_LIST_ITEM then list all modules wrapped in that macro.
print()
print("#ifdef MICROPY_FROZEN_LIST_ITEM")
for cm in compiled_modules:
    module_name = cm.source_file.str
    if module_name.endswith("/__init__.py"):
        short_name = module_name[: -len("/__init__.py")]
    else:
        short_name = module_name[: -len(".py")]
    print('MICROPY_FROZEN_LIST_ITEM("%s", "%s")' % (short_name, module_name))
print("#endif")
```

Picotool “Binary Information”

- Seems too easy of a starting point ;)
  - Other platforms may not encode this data
  - It’s not used by MicroPython itself, so we can “evade” this analysis by not letting it get built in (comment out the code in the previous slide, build, and…)

```
wes@thinkpad:~/pico/mp_redacted_info$ picotool info -a firmware.bin
File firmware.bin:

Program Information
 name:              MicroPython
 version:           v1.24.0-preview.36.g779ed0ace.dirty
 features:          thread support
                    USB REPL
 binary start:      0x10000000
 binary end:        0x10050aa4
 embedded drive:    0x100a0000-0x10200000 (1408K): MicroPython
```

Interacting with the REPL (if you can get it)

- tools/pytool.py
  - Execute commands
    - *./pyboard.py -c "help('modules')”*
      - A more reliable way of getting a list of modules (including frozen)
  - Filesystem manipulation
    - Listing
    - Uploading
    - Downloading
- tools/mpremote/mpremote.py
  - Similar functionality

Different Flavors of Hello World

helloworld.py > ...

```
1    import time
2
3    def go():
4        while True:
5            print("Hello Cruel World")
6            time.sleep(5)
```

- As a plaintext *.py* file (on the filesystem)
- As a compiled bytecode file (*.mpy*) (on the filesystem)
- As a frozen module compiled into MicroPython

...as a plaintext *.py* file (on the filesystem)

```
>>> wes@thinkpad:~/pico/helloworld$ ~/pico/micropython/tools/mpremote/mpremote.py fs cp helloworld.py :/helloworld.py
cp helloworld.py :/helloworld.py
```

```
wes@thinkpad:~/pico/helloworld$ ~/pico/micropython/tools/mpremote/mpremote.py repl
Connected to MicroPython at /dev/ttyACM0
Use Ctrl-] or Ctrl-x to exit this shell

>>> import helloworld
>>> helloworld.go()
Hello Cruel World
Hello Cruel World
```

As a compiled .mpy

```
d$ ~/pico/micropython/mpy-cross/build/mpy-cross helloworld.py
```

```
Address    00  01  02  03  04  05  06  07   08  09  0A  0B  0C  0D  0E  0F   ASCII

00000000:  4D  06  00  1F  08  01  1A  68   65  6C  6C  6F  77  6F  72  6C   M......helloworl
00000010:  64  2E  70  79  00  0F  08  74   69  6D  65  00  2B  04  67  6F   d.py...time.+.go
00000020:  00  0A  73  6C  65  65  70  00   2F  81  77  05  11  48  65  6C   ..sleep./.w..Hel
00000030:  6C  6F  20  43  72  75  65  6C   20  57  6F  72  6C  64  00  81   lo Cruel World..
00000040:  7C  08  0A  01  46  64  40  27   80  51  1B  02  16  02  32  00   |...Fd@'.Q....2.
00000050:  16  04  11  06  10  03  D9  44   45  11  04  34  00  59  51  63   .......DE..4.YQc
00000060:  01  81  48  10  08  04  60  20   27  12  07  23  00  34  01  59   ..H...` '..#.4.Y
00000070:  12  02  14  05  85  36  01  59   42  2F  51  63                    .....6.YB/Qc
```

```
Connected to MicroPython at /dev/ttyACM0
Use Ctrl-] or Ctrl-x to exit this shell

>>> import helloworld
>>> helloworld.go()
Hello Cruel World
Hello Cruel World
```

As a frozen bytecode module

ports > rp2 > boards > manifest.py

```
1  freeze("$(PORT_DIR)/modules")
2  include("$(MPY_DIR)/extmod/asyncio")
3  require("onewire")
4  require("ds18x20")
5  require("dht")
6  require("neopixel")
7  freeze("/home/wes/pico/helloworld","helloworld.mpy")
```

```
wes@thinkpad:~/pico/micropython/ports/rp2/build-RPI_PICO$ sudo ~wes/bin/picotool load firmware.uf2
Loading into Flash: [==============================]  100%
wes@thinkpad:~/pico/micropython/ports/rp2/build-RPI_PICO$ ~/pico/micropython/tools/mpremote/mpremote.py repl
Connected to MicroPython at /dev/ttyACM0
Use Ctrl-] or Ctrl-x to exit this shell
MicroPython v1.24.0-preview.37.gc12ad35f6.dirty on 2024-06-10; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> import helloworld
>>> helloworld.go()
Hello Cruel World
```

As a frozen bytecode module

ports > rp2 > boards > manifest.py

```
1  freeze("$(PORT_DIR)/modules")
2  include("$(MPY_DIR)/extmod/asyncio")
3  require("onewire")
4  require("ds18x20")
5  require("dht")
6  require("neopixel")
7  freeze("/home/wes/pico/helloworld","helloworld.mpy")
```

```
wes@thinkpad:~/pico/micropython/ports/rp2/build-RPI_PICO$ sudo ~wes/bin/picotool load firmware.uf2
Loading into Flash: [==============================]  100%
wes@thinkpad:~/pico/micropython/ports/rp2/build-RPI_PICO$ ~/pico/micropython/tools/mpremote/mpremote.py repl
Connected to MicroPython at /dev/ttyACM0
Use Ctrl-] or Ctrl-x to exit this shell
MicroPython v1.24.0-preview.37.gc12ad35f6.dirty on 2024-06-10; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> import helloworld
>>> helloworld.go()
Hello Cruel World
```

Rewind a bit. What can we learn from a .mpy?

- MicroPython's *tools/mpy-tool.py* is used in the build process to "freeze" .mpy's
  - Generates C code that's statically linked into the MicroPython firmware along with the other frozen modules
- Same tool also can dump useful info about a .mpy file with the disassemble flag *-d*

String Table

```
wes@thinkpad:~/pico/helloworld$ ../micropython/tools/mpy-tool.py -d helloworld.mpy
mpy_source_file: helloworld.mpy
source_file: helloworld.py
header: 4d:06:00:1f
qstr_table[8]:
    helloworld.py
    <module>
    time
    __main__
    go
    sleep
    __name__
    print
obj_table: ['Hello Cruel World']
```

Module Constructor

```
simple_name: <module>
  raw bytecode: 31 08:0a:01:46:64:40:27:80:51:1b:02:16:02:32:00:16:04:11:06:10:03:d9:44:45:11:04:34:00:59:51:63
  prelude: (2, 0, 0, 0, 0, 0)
  args: []
  line info: 46:64:40:27
  80          LOAD_CONST_SMALL_INT 0
  51          LOAD_CONST_NONE
  1b:02       IMPORT_NAME time
  16:02       STORE_NAME time
  32:00       MAKE_FUNCTION 0
  16:04       STORE_NAME go
  11:06       LOAD_NAME __name__
  10:03       LOAD_CONST_STRING __main__
  d9          BINARY_OP 2 __eq__
  44:45       POP_JUMP_IF_FALSE 5
  11:04       LOAD_NAME go
  34:00       CALL_FUNCTION 0
  59          POP_TOP
  51          LOAD_CONST_NONE
  63          RETURN_VALUE
  children: ['go']
```

“go” function

```
simple_name: go
  raw bytecode: 25 10:08:04:60:20:27:12:07:23:00:34:01:59:12:02:14:05:85:36:01:59:42:2f:51:63
  prelude: (3, 0, 0, 0, 0, 0)
  args: []
  line info: 60:20:27
  12:07       LOAD_GLOBAL print
  23:00       LOAD_CONST_OBJ 'Hello Cruel World'
  34:01       CALL_FUNCTION 1
  59          POP_TOP
  12:02       LOAD_GLOBAL time
  14:05       LOAD_METHOD sleep
  85          LOAD_CONST_SMALL_INT 5
  36:01       CALL_METHOD 1
  59          POP_TOP
  42:2f       JUMP -17
  51          LOAD_CONST_NONE
  63          RETURN_VALUE
  children: []
```

We can also get a roughly annotated hex dump with *-x*

```
wes@thinkpad:~/pico/helloworld$ ../micropython/tools/mpy-tool.py -x helloworld.mpy
00000000: 4d06 001f 0801 1a68 656c 6c6f 776f 726c  M......helloworl  helloworld.py
00000010: 642e 7079 000f 0874 696d 6500 2b04 676f  d.py...time.+.go  <module> time __main__ go
00000020: 000a 736c 6565 7000 2f81 7705 1148 656c  ..sleep./.w..Hel  sleep __name__ print Hello Cruel World
00000030: 6c6f 2043 7275 656c 2057 6f72 6c64 0081  lo Cruel World..
00000040: 7c08 0a01 4664 4027 8051 1b02 1602 3200  |...Fd@'.Q....2.  <module>
00000050: 1604 1106 1003 d944 4511 0434 0059 5163  .......DE..4.YQc
00000060: 0181 4810 0804 6020 2712 0723 0034 0159  ..H...` '..#.4.Y  go
00000070: 1202 1405 8536 0159 422f 5163            .....6.YB/Qc
```

So…

- If we can rebuild a .mpy of a frozen module out of information we extract from firmware, we have something we can use to inspect that.
  - It'll take a little RE work, as it's not in there as a contiguous binary blob.
  - It's in a set of structures that we'll have to locate/walk.
- We'll need to understand MicroPython QSTR, "string interning"
- We'll take a look at the bytecode
  - It's not compatible with CPython ("desktop" python) bytecode
- We can do some analysis with the "Unix" MicroPython build.

What's String Interning?

- Avoids duplicate copies of identical strings, especially identifiers
- **QSTR** : uni**Q**ue **STR**ing
- **QSTR values**: index into a linked list of **QSTR pools**
- References:
  - https://docs.micropython.org/en/latest/develop/qstr.html
  - https://en.wikipedia.org/wiki/String_interning
- There's a list of the static ones in *pi/makeqstrdata.py*

```
# static qstrs, these must ma
# See QSTR_LAST_STATIC at the

static_qstr_list = [
    "",
    "__dir__",  # Put __dir__
    "\n",
    " ",
    "*",
    "/",
    "<module>",
    "_",
    "__call__",
    "__class__",
    "__delitem__",
    "__enter__",
    "__exit__",
    "__getattr__",
    "__getitem__",
```

String interning in frozen code generated by mpy-tool.py

```
enum {
    MP_QSTR_Hello_space_Cruel_space_World = MP_QSTRnumber_of,
    MP_QSTR_go,
    MP_QSTR_helloworld_dot_py,
    MP_QSTR_sleep,
    MP_QSTR_time,
};

const qstr_hash_t mp_qstr_frozen_const_hashes[] = {
    136,
    109,
    194,
    234,
    240,
};

const qstr_len_t mp_qstr_frozen_const_lengths[] = {
    17,
    2,
    13,
    5,
    4,
};
```

```
extern const qstr_pool_t mp_qstr_const_pool;
const qstr_pool_t mp_qstr_frozen_const_pool = {
    &mp_qstr_const_pool, // previous pool
    MP_QSTRnumber_of, // previous pool size
    true, // is_sorted
    5, // allocated entries
    5, // used entries
    (qstr_hash_t *)mp_qstr_frozen_const_hashes,
    (qstr_len_t *)mp_qstr_frozen_const_lengths,
    {
        "Hello Cruel World",
        "go",
        "helloworld.py",
        "sleep",
        "time",
    },
};
```

String interning in frozen code generated by mpy-tool.py

```
static const qstr_short_t const_qstr_table_data_helloworld[8] = {
    MP_QSTR_helloworld_dot_py,
    MP_QSTR__lt_module_gt_,
    MP_QSTR_time,
    MP_QSTR___main__,
    MP_QSTR_go,
    MP_QSTR_sleep,
    MP_QSTR___name__,
    MP_QSTR_print,
};
```

Finding the module in a compiled firmware

- Ghidra!
  - ARM:LE:32:v8:default (for Pico)
- We can get our bearings in a debug build (.*elf* builds with frozen modules we know about)
- In an extracted/.bin firmware we won't have symbols, so we'll need to figure out how to find the data structures we need if we want to do this for any frozen module.
  - Load at base address 0x10000000

Debug Build – mp_frozen_names

```
                        s_p2.py_10046f85                        XREF[8,2]:   Entry Point (*),
                        mp_frozen_names                                      mp_builtin_help:100016be (R),
                                                                             mp_builtin_help:100016ee (*),
                                                                             mp_builtin_help:100016f8 (*),
                                                                             mp_find_frozen_module:10008f0e ...
                                                                             mp_find_frozen_module:10008f14 ...
                                                                             mp_find_frozen_module:10008f22 ...
                                                                             mp_find_frozen_module:10008f4c ...
                                                                             mp_builtin_help:100016be (*),
                                                                             mp_find_frozen_module:10008f0e ...
                                                                                frozen_content.c:13621
10046f84  72 70 32        ds          "rp2.py"
          2e 70 79
          00
10046f8b  5f 62 6f        ds          "_boot.py"
          6f 74 2e
          70 79 00
10046f94  5f 62 6f        ds          "_boot_fat.py"
          6f 74 5f
          66 61 74
```

```
          00
10047032  6e 65 6f        ds          "neopixel.py"
          70 69 78
          65 6c 2e ...
1004703e  68 65 6c        ds          "helloworld.py"
          6c 6f 77
          6f 72 6c ...
1004704c  00              ??          00h
1004704d  00              ??          00h
```

Debug Build – mp_frozen_mpy_content

```
                    mp_frozen_mpy_content                          XREF[2]:     Entry Point
                                                                                mp_find_fro
   10046f48  a8 90 04        mp_froze...
             10 10 90
             04 10 8c ...
     10046f48  a8 90 04 10    mp_frozen_module_t *   frozen_module_rp2              [0]

     10046f4c  10 90 04 10    mp_frozen_module_t *   frozen_module__boot            [1]
     10046f50  8c 8f 04 10    mp_frozen_module_t *   frozen_module__boot_fat        [2]
     10046f54  98 8e 04 10    mp_frozen_module_t *   frozen_module_asyncio___init__ [3]
     10046f58  08 87 04 10    mp_frozen_module_t *   frozen_module_asyncio_core     [4]
     10046f5c  2c 85 04 10    mp_frozen_module_t *   frozen_module_asyncio_event    [5]
     10046f60  f4 81 04 10    mp_frozen_module_t *   frozen_module_asyncio_funcs    [6]
     10046f64  68 80 04 10    mp_frozen_module_t *   frozen_module_asyncio_lock     [7]
     10046f68  8c 7a 04 10    mp_frozen_module_t *   frozen_module_asyncio_stream   [8]
     10046f6c  48 7a 04 10    mp_frozen_module_t *   frozen_module_uasyncio         [9]
     10046f70  58 77 04 10    mp_frozen_module_t *   frozen_module_onewire          [10]
     10046f74  20 75 04 10    mp_frozen_module_t *   frozen_module_ds18x20          [11]
     10046f78  08 73 04 10    mp_frozen_module_t *   frozen_module_dht              [12]
     10046f7c  b8 70 04 10    mp_frozen_module_t *   frozen_module_neopixel         [13]
     10046f80  50 70 04 10    mp_frozen_module_t *   frozen_module_helloworld       [14]
```

Finding them in the non-debug build

- Eyeball the name table: Look for the list of known frozen modules (such as “asyncio/__init__.py”, etc.), this is **mp_frozen_names**
- Code reference from a function that uses it indirectly, this is **mp_find_frozen_module**:

```
                    LAB_10008f08                                    XREF[1]:     10008f02 (j)
10008f08  00 23         movs        r3,#0x0
10008f0a  16 4d         ldr         r5,[DAT_10008f64 ]                          = 10046F84h
10008f0c  01 93         str         r3,[sp,#local_24 ]

                    LAB_10008f0e                                    XREF[1]:     10008f5c (j)
10008f0e  28 78         ldrb        r0,[r5,#0x0]=>DAT_10046f84                  = 72h
                                                                                = "p2.py"
```

```
                    DAT_10008f64
10008f64  84 6f 04        undefine ... 10046F84h
          10
```

Finding them in the non-debug build

- Later in **mp_find_frozen_module** there is an indirect reference to **mp_frozen_mpy_content**

```
10008f3c  0a 4b        ldr       r3,[DAT_10008f68 ]
10008f3e  92 00        lsls      r2,r2,#0x2
10008f40  d3 58        ldr       r3=>PTR_LAB_10046f48 ,[r2,r3]
```

```
                        DAT_10008f68
10008f68  48 6f 04 10       undefine ...  10046F48h
```

Reconstructing **helloworld.mpy**

- Header: 4D 06 00 1F
  - 0x4D - ‘M’
  - 0x06 - major MicroPython version
  - 0x00 - non-native bytecode
  - 0x1F – number of bits in a small int
- **vuint** – variably encoded unsigned integer
  - 7 bits per byte
  - 8th bit (most significant bit) set if another byte follows

https://docs.micropython.org/en/latest/reference/mpyfiles.html

Reconstructing **helloworld.mpy**

size	field	
vuint	number of qstrs	**0x08**
vuint	number of constant objects	**0x01**
...	qstr data	
...	encoded constant objects	

```
                                const_qstr_table_data_helloworld
10047060  38 04           ushort      1080
10047062  07 00           ushort      7
10047064  8b 03           ushort      907
10047066  15 00           ushort      21
10047068  36 04           ushort      1078
1004706a  5f 03           ushort      863
1004706c  17 00           ushort      23
1004706e  7b 00           ushort      123

                                const_obj_table_data_helloworld
1004705c  52 1e 00        uint        1E52h
          00
```

These are written out by mpy-tool as MP_ROM_* types, check the C source definitions to figure out how to tell them apart.

```
#define MP_OBJ_NEW_QSTR(qst) ((mp_obj_t)((((mp_uint_t)(qst)) << 3) | 2))
```

Reconstructing **helloworld.mpy**

- **qstr data**
  - Frozen modules' QSTRs get mixed together in **mp_qstr_frozen_const_pool**
  - To find that, either
    - Follow *main* > *mp_init* > *qstr_init,* (referenced in the first four instructions)
    - Search for a string that you know is part of a frozen module and reference back

https://docs.micropython.org/en/latest/reference/mpyfiles.html

Reconstructing **helloworld.mpy**

```
typedef struct _qstr_pool_t {
    const struct _qstr_pool_t *prev;
    size_t total_prev_len : (8 * sizeof(size_t) - 1);
    size_t is_sorted : 1;
    size_t alloc;
    size_t len;
    #if MICROPY_QSTR_BYTES_IN_HASH
    qstr_hash_t *hashes;
    #endif
    qstr_len_t *lengths;
    const char *qstrs[];
} qstr_pool_t;
```

https://docs.micropython.org/en/latest/reference/mpyfiles.html

Reconstructing **helloworld.mpy**

- **qstr data**
  - Previous pool is **mp_qstr_const_pool**
  - **total_prev_len** helps you see which pool you need to consult (0x3C2 in our example)
    - So any QSTR index > 962, is in this frozen pool
    - Less than, we'll have to consult backwards into previous pools.

```
                        mp_qstr_frozen_const_pool                  XREF[3]:   FUN_100174f0:1001
                                                                              FUN_10017828:1001
                                                                              10017878 (*)
100499ec e0 0c 04 10    addr        DAT_10040ce0                               = 70h     p
100499f0 c2 03 00 80    ddw         800003C2h
100499f4 0a 00 00 00    ddw         Ah
100499f8 eb 00 00 00    ddw         EBh
100499fc 9b 9e 04 10    addr        frozen_hashes                              = 14h
10049a00 b0 9d 04 10    addr        frozen_lengths                             = 05h
10049a04 09 d3 03 10    addr        DAT_1003d309                               = 30h     0
10049a08 0f d3 03 10    addr        s_CRC_error_1003d30f                       = "CRC error"
```

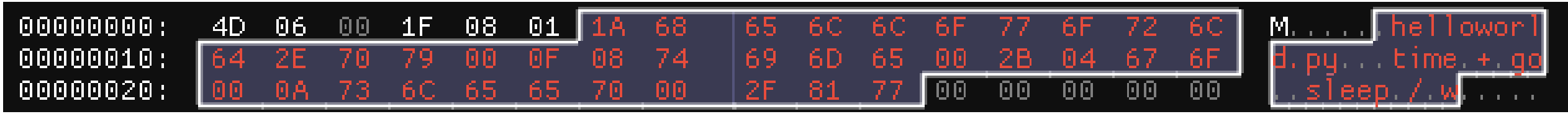
https://docs.micropython.org/en/latest/reference/mpyfiles.html

Reconstructing **helloworld.mpy**

```
                        const_qstr_table_data_helloworld
10047060  38 04           ushort      1080    1080-962 = 118 ← “helloworld.py”
10047062  07 00           ushort      7    mp_qstr_const_pool_static > “<module>”
10047064  8b 03           ushort      907  mp_qstr_const_pool > “time”
10047066  15 00           ushort      21   mp_qstr_const_pool_static > “__main__”
10047068  36 04           ushort      1078   “go”
1004706a  5f 03           ushort      863  mp_qstr_const_pool > “sleep”
1004706c  17 00           ushort      23   mp_qstr_const_pool_static > “__name__”
1004706e  7b 00           ushort      123  mp_qstr_const_pool_static > “print”
```

Reconstructing **helloworld.mpy**

- .mpy string encoding:
  - First vuint: Low order bit determines if it's a static qstr entry or one defined in this .mpy
    - Static: Rest of the bits of this vuint are an index into static_qstr_list (conveniently in py/makeqstrdata.py
    - Defined in this .mpy: Rest of the bits of this vuint are a length for the string
  - String follows



Reconstructing **helloworld.mpy**

- Encoded constant objects area can get more complicated (dig into *mpy-tool.py*'s "read_obj" function for insight)
- Also check the MP_OBJ_* defines to figure out how to decode the object

```
                        const_obj_table_data_helloworld                 XREF[1]:     10047054 (*)
1004705c  52 1e 00 00             ddw       1E52h
```

```
#define MP_OBJ_NEW_QSTR(qst) ((mp_obj_t)((((mp_uint_t)(qst)) << 3) | 2))
```

Shifted back right by 3 bits, this is 0x3CA (970) - QSTR "Hello Cruel World"
We encode with MP_PERSISTENT_OBJ_STR, a length, and the string

```
Address     00  01  02  03  04  05  06  07   08  09  0A  0B  0C  0D  0E  0F   ASCII

00000000:   4D  06  00  1F  08  01  1A  68   65  6C  6C  6F  77  6F  72  6C   M.....helloworl
00000010:   64  2E  70  79  00  0F  08  74   69  6D  65  00  2B  04  67  6F   d.py...time.+.go
00000020:   00  0A  73  6C  65  65  70  00   2F  81  77  05  11  48  65  6C   ..sleep./.w..Hel
00000030:   6C  6F  20  43  72  75  65  6C   20  57  6F  72  6C  64  00  00   lo Cruel World..
```

Reconstructing **helloworld.mpy**

- Finally, the bytecode. Check out *mpy-tool.py*'s read_raw_code for the format we have to generate.
- Pointer to the code is the third field of the module, which then has a pointer to code and a pointer to a list of children
- This is what it looks like in debug build, for illustration:

```
                    frozen_module_helloworld                    XREF[1]:    10046f80(*)
10047050 60 70 04 10 5c 70        mp_frozen_module_t
         04 10 70 70 04 10
  10047050 60 70 04 10 5c mp_module_constants_t                          constants
           70 04 10
    10047050 60 70 04 10    qstr_short_t *        const_qstr_table_data_helloworld    qstr_table
    10047054 5c 70 04 10    mp_obj_t *            const_obj_table_data_helloworld     obj_table
  10047058 70 70 04 10    void *                proto_fun_helloworld__lt_module_gt_  proto_fun
```

Reconstructing **helloworld.mpy**

```
                        proto_fun_helloworld__lt_module_gt_              XREF[1]:     10047058 (*)
10047070  00 00 02 00 99 70        mp_raw_code_truncated ...
          04 10 7c 70 04 10
  10047070  00 00          uint8_t[2]          ""                                  proto_fun_indicator
    10047070  [0]            00h, 00h
  10047072  02             uint8_t             02h                                 kind
  10047073  00             _Bool               FALSE                               is_generator
  10047074  99 70 04 10    void *              fun_data_helloworld__lt_module_gt_  fun_data
  10047078  7c 70 04 10    _mp_raw_code_t * *  children_helloworld__lt_module_gt_  children
```

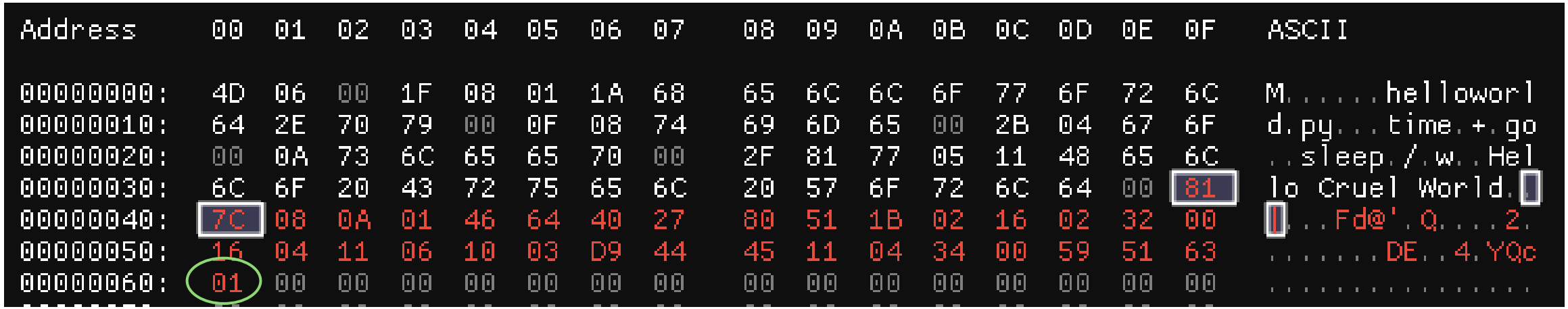
```
                        fun_data_helloworld__lt_module_gt_
10047099  08 0a 01 46 64 40        byte[31]
          27 80 51 1b 02 16
          02 32 00 16 04 11 ...
  10047099  [0]           '\b','\n', 01h, 'F',
  1004709d  [4]            'd', '@','\'', 80h,
  100470a1  [8]            'Q', 1Bh, 02h, 16h,
  100470a5  [12]           02h, '2', 00h, 16h,
  100470a9  [16]           04h, 11h, 06h, 10h,
  100470ad  [20]           03h, D9h, 'D', 'E',
  100470b1  [24]           11h, 04h, '4', 00h,
  100470b5  [28]           'Y', 'Q', 'c'
```

```
                        children_helloworld__lt_module_gt_          XREF[1]:
1004707c  80 70 04 10            mp_raw_code_t *[1]
  1004707c  80 70 04 10    mp_raw_code_t *     fun_data_helloworld_go

                        fun_data_helloworld_go                      XREF[1]:
10047080  10 08 04 60 20 27        byte[25]                    10h,"\b",04h,"` '",1
          12 07 23 00 34 01
          59 12 02 14 05 85 ...
  10047080  [0]           10h,'\b', 04h, '`',
  10047084  [4]           ' ','\'', 12h,'\a',
  10047088  [8]           '#', 00h, '4', 01h,
  1004708c  [12]          'Y', 12h, 02h, 14h,
  10047090  [16]          05h, 85h, '6', 01h,
  10047094  [20]          'Y', 'B', '/', 'Q',
  10047098  [24]          'c'
```

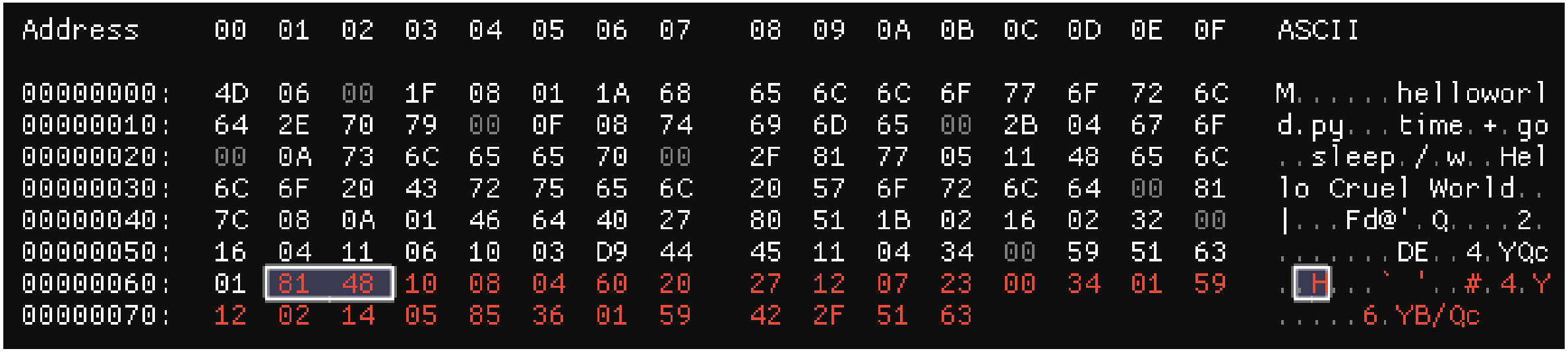
Reconstructing **helloworld.mpy**

- vuint for kind, length, and whether or not the code has children
  - Least significant 2 bits: “kind” – Basically if it’s bytecode or native code.
    - Here, 0 which will translate to MP_CODE_BYTECODE
  - Third bit: children? 1 for the first block of code, 0 for the second
  - Remaining bits: Length
- Length of code and number of children is tricky in stripped binary build: Use references as cues
- Each block of code followed recursively by its children



```
Address    00  01  02  03  04  05  06  07    08  09  0A  0B  0C  0D  0E  0F   ASCII

00000000:  4D  06  00  1F  08  01  1A  68    65  6C  6C  6F  77  6F  72  6C   M.....hellowor
00000010:  64  2E  70  79  00  0F  08  74    69  6D  65  00  2B  04  67  6F   d.py...time.+.go
00000020:  00  0A  73  6C  65  65  70  00    2F  81  77  05  11  48  65  6C   ..sleep./.w..Hel
00000030:  6C  6F  20  43  72  75  65  6C    20  57  6F  72  6C  64  00  81   lo Cruel World..
00000040:  7C  08  0A  01  46  64  40  27    80  51  1B  02  16  02  32  00   |...Fd@'.Q....2.
00000050:  16  04  11  06  10  03  D9  44    45  11  04  34  00  59  51  63   .......DE..4.YQc
00000060:  01  00  00  00  00  00  00  00    00  00  00  00  00  00  00  00   ................
```



```
Address    00  01  02  03  04  05  06  07    08  09  0A  0B  0C  0D  0E  0F   ASCII

00000000:  4D  06  00  1F  08  01  1A  68    65  6C  6C  6F  77  6F  72  6C   M.....hellowor
00000010:  64  2E  70  79  00  0F  08  74    69  6D  65  00  2B  04  67  6F   d.py...time.+.go
00000020:  00  0A  73  6C  65  65  70  00    2F  81  77  05  11  48  65  6C   ..sleep./.w..Hel
00000030:  6C  6F  20  43  72  75  65  6C    20  57  6F  72  6C  64  00  81   lo Cruel World..
00000040:  7C  08  0A  01  46  64  40  27    80  51  1B  02  16  02  32  00   |...Fd@'.Q....2.
00000050:  16  04  11  06  10  03  D9  44    45  11  04  34  00  59  51  63   .......DE..4.YQc
00000060:  01  81  48  10  08  04  60  20    27  12  07  23  00  34  01  59   ..H...` '..#.4.Y
00000070:  12  02  14  05  85  36  01  59    42  2F  51  63                   .....6.YB/Qc
```

Using the Unix port to examine an extracted module

```
wes@thinkpad:~/pico/helloworld/extracted$ ../../micropython/ports/unix/build-standard/micropython
MicroPython v1.24.0-preview.37.gc12ad35f6.dirty on 2024-06-11; linux [GCC 13.2.0] version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>> import helloworld
>>> import micropython
>>> micropython.qstr_info(True)
qstr pool: n_pool=1, n_qstr=5, n_str_data_bytes=64, n_total_bytes=352
Q(/usr/lib/micropython)
Q(helloworld)
Q(helloworld.mpy)
Q(helloworld.py)
Q(go)
```

```
>>> helloworld.__dict__
{'time': <module 'time'>, 'go': <function go at 0x7fa56baca740>, '__na
me__': 'helloworld', '__file__': 'helloworld.mpy'}
```

```
>>> helloworld.go()
Hello Cruel World
```

lightemup.mpy

```
  children: ['doit']
simple_name: doit
  raw bytecode: 132 50:20:05:60:20:23:25:5a:22:20:26:25:29:27:
02:b1:10:07:12:02:13:08:34:82:01:36:01:59:81:e5:57:9d:d7:43:25
:55:14:09:80:36:01:59:b1:b2:f8:b3:d9:44:49:b0:b1:55:14:09:81:3
0:36:01:59:81:e5:58:5a:d7:43:08:59:59:b2:81:e5:c2:b2:87:d8:44:
  prelude: (11, 0, 0, 0, 0, 0)
  args: []
  line info: 60:20:23:25:5a:22:20:26:25:29:27:31:32:24:25
  2b:00       BUILD_LIST 0
  c0          STORE_FAST 0
  80          LOAD_CONST_SMALL_INT 0
  42:56       JUMP 22
  57          DUP_TOP
  c1          STORE_FAST 1
  b0          LOAD_FAST 0
  14:06       LOAD_METHOD append
  12:02       LOAD_GLOBAL Pin
  b1          LOAD_FAST 1
  10:07       LOAD_CONST_STRING mode
  12:02       LOAD_GLOBAL Pin
  13:08       LOAD_ATTR OUT
  34:82:01    CALL_FUNCTION 257
  36:01       CALL_METHOD 1
  59          POP_TOP
  81          LOAD_CONST_SMALL_INT 1
  e5          BINARY_OP 14 __iadd__
```

Our own machine.py that implements Pin

```python
class Pin():
    IN = "in"
    OUT = "out"
    n = 0
    def __init__(self, no, mode=IN):
        self.n = no
    def value(self, v=None):
        print('Pin ' + str(self.n) + ': ' + str(v))
    def deinit(self):
        return
```

Now we can run it in the Unix port

```
wes@thinkpad:~/pico/lightemup/extracted$ ../../micropy
thon/ports/unix/build-standard/micropython
MicroPython v1.24.0-preview.37.gc12ad35f6.dirty on 202
4-06-11; linux [GCC 13.2.0] version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>> import machine
>>> import lightemup
>>> lightemup.doit()
Pin 0: 0
Pin 0: 1
Pin 1: 0
Pin 1: 1
Pin 2: 0
```

-DMICROPY_PY_SYS_SETTRACE

https://forum.micropython.org/viewtopic.php?t=4704

```
MicroPython v1.24.0-preview.37.gc12ad35f6.dirty on 2024-06-11; linux [GCC 13.2.0] version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>> import sys
>>> import machine
>>> import lightemup
>>> def trace(frame, event, arg):
...     print(event, frame.f_code.co_name, frame.f_code.co_code)
...
>>> sys.settrace(trace)
>>> lightemup.doit()
call <module> b'\x11\x03\x11\x04\x14\x026\x004\x01YQc'
call doit b'+\x00\xc0\x80BVW\xc1\xb0\x14\x06\x12\x02\xb1\
d7C%Y\x80\xc2\xb2\x80BsW\xc3\x80B]W\xc1\xb0\xb1U\x14\t\x8
Y\x81\xe5W\x9d\xd7C\x1eY\x12\x04\x14\n#\x006\x01Y\x81\xe5
xb4\x7fQc'
call __init__ b'\xb1\xb0\x18\x06Qc'
call __init__ b'\xb1\xb0\x18\x06Qc'
```

```
call __init__ b'\xb1\xb0\x18\x06Qc'
call __init__ b'\xb1\xb0\x18\x06Qc'
call __init__ b'\xb1\xb0\x18\x06Qc'
call value b'\x12\x14\x10\x08\x12\x15\xb0\x13\x064
Pin 0: 0
call value b'\x12\x14\x10\x08\x12\x15\xb0\x13\x064
Pin 0: 1
call value b'\x12\x14\x10\x08\x12\x15\xb0\x13\x064
Pin 1: 0
call value b'\x12\x14\x10\x08\x12\x15\xb0\x13\x064
Pin 1: 1
call value b'\x12\x14\x10\x08\x12\x15\xb0\x13\x064
Pin 2: 0
call value b'\x12\x14\x10\x08\x12\x15\xb0\x13\x064
```

Understanding the Bytecode

```
simple_name: go
  raw bytecode: 25 10:08:04:60:20:27:12:07:23:00:34:01:59:12:02:14:05:85:36:01:59:42:2f:51:63
  prelude: (3, 0, 0, 0, 0, 0)
  args: []
  line info: 60:20:27
  12:07         LOAD_GLOBAL print
  23:00         LOAD_CONST_OBJ 'Hello Cruel World'
  34:01         CALL_FUNCTION 1
  59            POP_TOP
  12:02         LOAD_GLOBAL time
  14:05         LOAD_METHOD sleep
  85            LOAD_CONST_SMALL_INT 5
  36:01         CALL_METHOD 1
  59            POP_TOP
  42:2f         JUMP -17
  51            LOAD_CONST_NONE
  63            RETURN_VALUE
```

- References – Source diving!
  - mpy-tool.py's "Opcode" class
  - py/vm.c
- Stack-based
- Not compatible with CPython bytecode, but familiarity with it will help

Conclusions

- I'd *really* rather be reversing compiled C
  - But it's doable :)
- Frozen modules are a great way to efficiently include project-specific code on a firmware image
  - But it is **not** a way to hide/obfuscate code or secrets
  - It's only as well-protected as access to your firmware (if you're using a microcontroller that supports that)
    - Dumping from flash
    - Dumping from REPL
    - Updates
- **This is not a layer of protection**
- Could automate the extraction process, but it's a bit tricky to determine lengths of some of the data structures without eyeballing the context.

Contact

Dr. Wesley McGrew
Senior Cyber Fellow
MartinFed

w.mcgrew@martinfed.com
@McGrewSecurity

