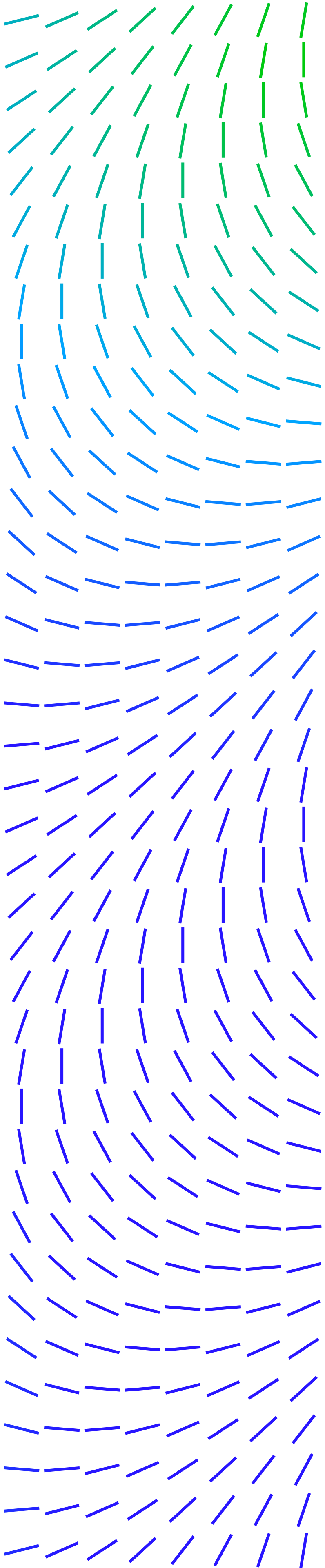


WHITE PAPER

Emulating Code with Unicorn

Trellix

Introduction

When analyzing a piece of malware, or reversing a CTF challenge, it's common to find functions that are implementing a given algorithm that you want to apply to arbitrary data. Common examples can be de-obfuscating strings, computing a CRC, decompressing a custom compression routine, etc. Potential ways to go about it is to either reverse the algorithm and re-implement it yourself, or maybe follow the YOLO approach and click around in a debugger, manually feeding data to the processing code. But what if there were an in-between, perhaps the ability to create a standalone script that performs the desired task without having to painstakingly reverse engineer the algorithm? What if there was something that could save us from the manual effort of running questionable code inside a debugger to directly execute the processing code? This mythical solution is real and is called Unicorn Engine. Based on QEMU, it will emulate the code you provide. With bindings to many different scripting languages, you can leverage its horsepower from your favorite scripting environment. In this Tools and Techniques article, we will be relying on Python 3 to solve some of these challenges. Buckle up buckaroos and let's go on a ride with this powerful tool.

Getting Started

You can see from the download webpage that you have many options to install Unicorn. The most straight forward is to run:

```
pip install unicorn
```

From there you can make sure everything works as expected by running a simple script such as:

```
from unicorn import *
from unicorn.x86_const import *

def init_mu():
    mu = Uc(UC_ARCH_X86, UC_MODE_32)  # Create the emulator in mode x86 (32bit)
    STACK = 0x3000000                 # Pick an arbitrary address for the stack base
    heap_base = 0x6000000             # Pick an arbitrary address for the heap base
    heap_size = 0x10000               # Pick an arbitrary size for the heap
    stack_size = 0x3000               # Pick an arbitrary size for the stack (whatever as long as it is "big enough")
    mu.mem_map(STACK, stack_size )    # Create memory mapping for the stack based on what we picked
    mu.mem_map(heap_base, heap_size)  # Create memory mapping for the heap based on what we picked
    mu.mem_map(0, 0x1000)             # Map address 0 as the "mov large:fs 0 , OFF_SEH_xxx" will dereference that

    mu.reg_write(UC_X86_REG_ESP, STACK+ stack_size -4 - 0x200)   # Adjust ESP and EBP based on our mapping and
    mu.reg_write(UC_X86_REG_EBP, STACK+ stack_size - 4 )         # make a stack frame of size 0x200 (for local variables)

    return mu
```

The code above initializes Unicorn in a such a way that it will be ready to execute x86-32 code. We have added some extra fluff such as pre-allocating space for heap and stack as eventually our goal is to execute whole functions.

We can now run the equivalent of a hello world:

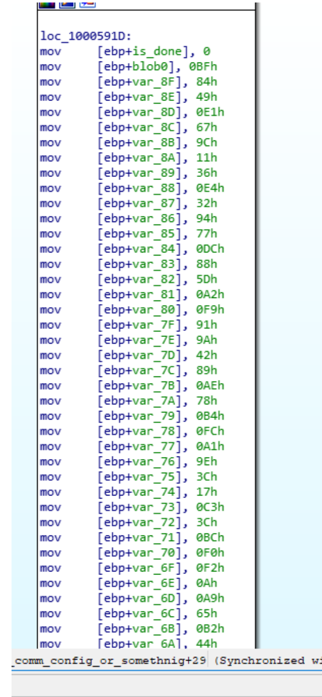
```
def hello_world():
    """
    xor eax, eax
    mov eax, 0x42
    add eax, 0x12f5
    """
    payload = b"\x31\xC0\xB8\x42\x00\x00\x00\x05\xF5\x12\x00\x00"
    mu = init_mu()

    ADDRESS = 0x10001000
    mu.mem_map(ADDRESS, 0x1000000)                # Reserve 0x1000000 bytes for the code at an arbitrary address
    mu.mem_write(ADDRESS,  payload)               # Load the program in the memory we mapped
    mu.reg_write(UC_X86_REG_EIP, ADDRESS)         # Set EIP to our address

    START_ADDRESS = ADDRESS
    END_ADDRESS = START_ADDRESS + len(payload)

    try:                                          # Boilerplate code to start the emulation
        # emulate machine code in infinite time
        mu.emu_start(START_ADDRESS, END_ADDRESS)
    except UcError as e:
        print("ERROR: %s" % e)
        print("at: %x" % mu.reg_read(UC_X86_REG_EIP))

    res = mu.reg_read(UC_X86_REG_EAX)             # Read EAX register after execution
    print(hex(res))                               # Prints the result (0x1337)
```

And as expected, we should see the following printed in the console:

```
0x1337
```

Emulating real life code

Emulating a Hello World is a good start, but what about a real-life scenario? The following screenshot is taken from a piece of malware:



The code above does something fairly standard with malware: it slowly builds onto the stack the blobs it is going to process. In this case, this is encrypted data that will require more processing, but more often than not, it might directly be strings. This is done in such a way to avoid detection by an antivirus and/or making it harder for a reverser to spot interesting strings directly in the binary.

In order to tackle this process efficiently, a good approach is as follow:

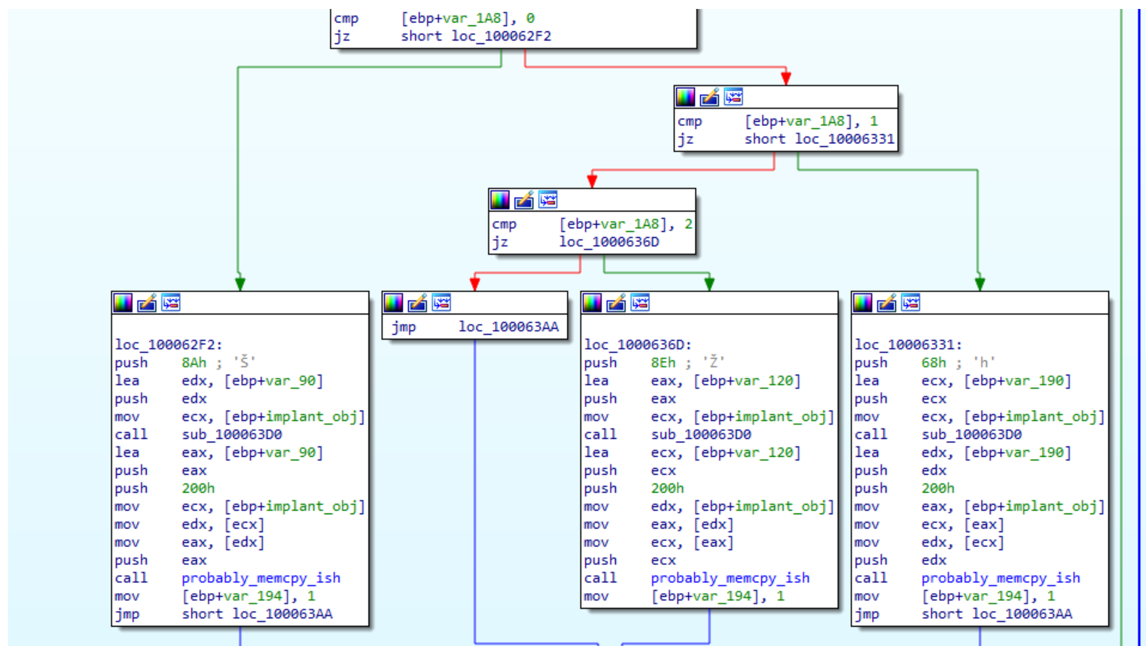
1. Load the binary at the same address as it is showing in your reversing tool
2. Make sure the stack is set up properly (as we do in the init_mu code):
   - Map a memory range for the stack
   - Set ESP/EBP accordingly
   - It is preferable to have EBP higher in memory than ESP to have a stack frame ready (EBP = ESP + 0x200 for example).
3. Set EIP to the beginning of the code you want to emulate.
4. Set the end of the emulation at the end of the code building the data on the stack. Note: if there are control flow changes (conditional jumps and function calls) this might lead unexpected complications, some addressed below.

5. Read the memory back from Unicorn and get the data.

Implemented in Python, it looks like this:

```
def emulate_stack_building(elf_program):
    elf_program = elf_program[0x400:]                   # Skip headers to start of .text section
    ADDRESS =     0x10001000                            # Load address of the .text section
    START_ADDRESS = 0x10005927                          # Address we want to emulate from Reversing tool
    END_ADDRESS =   0x1000625D                          # End address based on reversing tool

    mu = init_mu()
    mu.mem_map(ADDRESS, 0x1000000)
    mu.mem_write(ADDRESS,  elf_program)
    mu.reg_write(UC_X86_REG_EIP, START_ADDRESS)

    try:
        # emulate machine code in infinite time
        mu.emu_start(START_ADDRESS, END_ADDRESS)
    except UcError as e:
        print("ERROR: %s" % e)
        print("at: %x" % mu.reg_read(UC_X86_REG_EIP))


    ebp = mu.reg_read(UC_X86_REG_EBP)

    stack_data = mu.mem_read(ebp-0x200, 0x200)          # Read stack frame previously allocated
    config0 = stack_data[-0x90:-0x6]                    # Extract arrays config0, config1, config2
    config1 = stack_data[-0x190:-0x128]                 # Location and size is based on reversing.
    config2 = stack_data[-0x120:-0x92]                  # For example ebp+var_90 is really ebp-0x90,
                                                        # hence the negative offsets.
    return (config0, config1, config2), stack_data
```

If you are curious about the offsets chosen at the end of the function, this is because in our malware example, the data is split in three different blobs, which we replicated in the script. For the sake of completeness, here's a screenshot of the malware code doing the split:



Going Further

Emulating function calls

I'd like to share a couple of extra pointers to make life easier when it comes to applying this technique to a real-life scenario. Something you might encounter is a desire to emulate a function call. To do so, there are two things to keep in mind. First, you need to be aware of the relevant calling convention being used (i.e., do you need to push arguments onto the stack, or do you need to set appropriate registers?). This can be figured out by simply looking at the code invoking the function of interest and following the same pattern. The other important caveat is that if the function calls other functions, as the complexity grows, there is a greater risk of executing code that will try to access something not initialized and/or call libraries that are not mapped. We will discuss in the next section a potential way to mitigate some of this but be aware that complexity is not your friend in this scenario.

A useful trick when it comes to writing an emulation script that relies on invoking functions is to implement simple stack management to make code more legible. For instance, you can implement custom push/pop functions that will enable your code to operate at the proper layer of abstraction, as opposed to constantly managing the stack manually.

Here's how this can be done in python:

```
def push(mu, val):
    esp = mu.reg_read(UC_X86_REG_ESP)
    mu.mem_write(esp-4, struct.pack("<I", val))
    mu.reg_write(UC_X86_REG_ESP, esp - 4)
def pop(mu):
    esp = mu.reg_read(UC_X86_REG_ESP)
    val = struct.unpack("<I", mu.mem_read(esp,4))[0]
    mu.reg_write(UC_X86_REG_ESP, esp + 4)
    return val
```

The code above leverages the fact that the Unicorn engine "knows" the value of ESP and updates it accordingly. Here we assume a push/pop operates on DWORDs, and increment/decrement the stack by 4; this should be adapted for other architectures (e.g., push QWORD in x64).

Intercepting library calls

One of the limitations described previously is when the code is referencing shared libraries and other API calls. Short of re-implementing a PE/ELF loader and emulating the whole binary, a reasonable approach is to intercept these function calls in our script and re-implement the intended behavior or something close enough to serve our purpose. A common example would be heap management functions; for instance: malloc and free, and their equivalents.

First, we can implement a generic interception framework. To do so, we can either overwrite an existing instruction with the "breakpoint opcode" (on x86 that would be 0xCC) or just trace each instruction being executed, while keeping track in either case of which callback function we want to execute if the instruction is ever hit (when EIP == address). On hit, we execute the handler instead of the original code, and let the handler decide how to resume execution. For example, here's how we implement this in Python:

```
def get_arg_at_func_start(mu, pos):
    """ We assume this is called when the function starts, and only rip is on the stack before args"""
    esp = mu.reg_read(UC_X86_REG_ESP)
    val = struct.unpack("<I", mu.mem_read(esp+4*(pos+1), 4))[0]
    return val

def ret(mu):
    rip = pop(mu)
    print("Ret: " + hex(rip))
    mu.reg_write(UC_X86_REG_EIP, rip)

def on_malloc(mu, address):
    global heap_pos, heap_base, heap_size              # We assume the heap is already defined and mapped
    print("malloc")                                    # For debug
    size = get_arg_at_func_start(mu, 0)                # Get the first argument that was pushed on the stack
    mu.reg_write(UC_X86_REG_EAX, heap_pos)             # Save the result of the mem allocation in EAX (based on calling convention)
    heap_pos += size                                   # Keep track of memory we have allocated for next call
    print(hex(size))                                   # For debug
    ret(mu)                                            # Perform a regular return from the call.

breakpoint(mu,0x1000793F, on_malloc)                   # Example on how to use the code to intercept a malloc
    if insertBP:
        mu.mem_write(address, b"\xcc\x90\x90\x90")     # We write a 0xCC that will trigger a user interrupt.
                                                       # The \x90 are nops which may have side effects if
                                                       # the original instruction is shorter than 4 bytes...
                                                       # For a proper bp, we should also be ready to emulate/restore
                                                       # the original instruction, rather than just overwriting it.
    bp_dict[hex(address)] = handler
```

Then, leveraging this framework, we can implement basic heap management to replace malloc/free. It doesn't need to be robust; as long as we provide memory that we have mapped and is not being used elsewhere, we likely don't need to implement a full-fledged heap management framework with free lists and such. For example, here's how we could intercept malloc:

The code above keeps a pointer to the first byte of free memory we have allocated for the heap. On a call to malloc, it retrieves the desired size, sets EAX (the return value) to this memory address and then increments the free heap pointer with the desired amount of memory passed to malloc. This way, the next call to malloc will also receive a fresh chunk of memory. Then, the handler pops from the stack the return address of the function and updates EIP accordingly. This assumes the address we are intercepting is the first instruction of malloc (before it creates a stack frame and modifies ESP and EBP).

Conclusion

In this “Tools and Techniques” paper, we’ve seen how we can leverage the Unicorn engine to emulate code. This offers an opportunity to create standalone scripts that will process data based upon code that is not necessary to fully reverse. Some difficulties remain if the code is too complex, which can be partially mitigated by hooking library functions and re-implementing some simplified versions of these. With the framework described in this document, we were able to emulate and decrypt data used by a real-life malware and provide a standalone script to extract/decrypt its configuration without having to reverse the whole decryption algorithm.

Trellix

Copyright © 2022 Musarubra US LLC
JANUARY 2022