

1 Preface

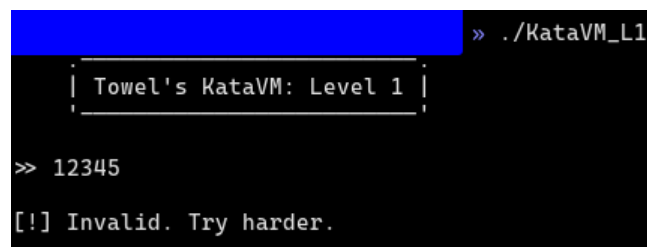
Greetings! In this article, I will reverse engineer [VM obfuscated crackme](#). During reverse engineering I tried several analysis techniques, some of them paid off, and some did not. I will write here about my struggles to analyze this machine using different tools. Consider opening [chapter 4](#) if you are interested only in VM solving because in the next 2 chapters I will mention different analysis techniques that have shown themselves useless for this case.

1.1 Toolset

- Radare2/Rizin - for dynamic and static analysis of assembly code (I use these tools interchangeably depend on which bugs don't bother me in given situation)
- Python - for writing VM bytecode disassembler. I tried several other specialized tools like miasm (it is possible to use it as a disassembler, not only a symbolic execution engine), and sleigh (encountered troubles compiling it). Other solutions like binja architecture plugins require paying money, so I will omit them. Based on the fact that this is not a long-run project, I decided to write my own bicycle rather than debugging compilation problems in eclipse.
- Cutter - for viewing graphs - some people prefer ghidra over the cutter, but on my machine ghidra GUI was unable to run smoothly, so I had no other options (Besides that - cutter graph view looks neat)
- Angr - It has broad documentation and easy installation steps, this is the reason i chose this symbolic execution framework.
- AFL++ - Same as above, fast installation and decent documentation.

1.2 First look

Let's launch crackme and see what we need to do:



```
» ./KataVM_L1
| Towel's KataVM: Level 1 |
» 12345
[!] Invalid. Try harder.
```

Figure 1: first run

So we need to provide right input to see some kind of "win" message.

2 Virtual machine

The first thing I did is checked this file with [DIE](#) - maybe I am dealing with some common type of VM.

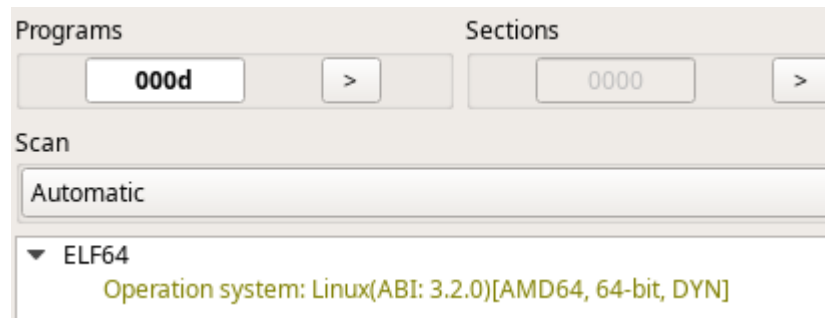


Figure 2: results

Well, looks like I am working with a makeshift obfuscation. Let's take a look at architecture of most VMs to get the general idea of what we are working with:

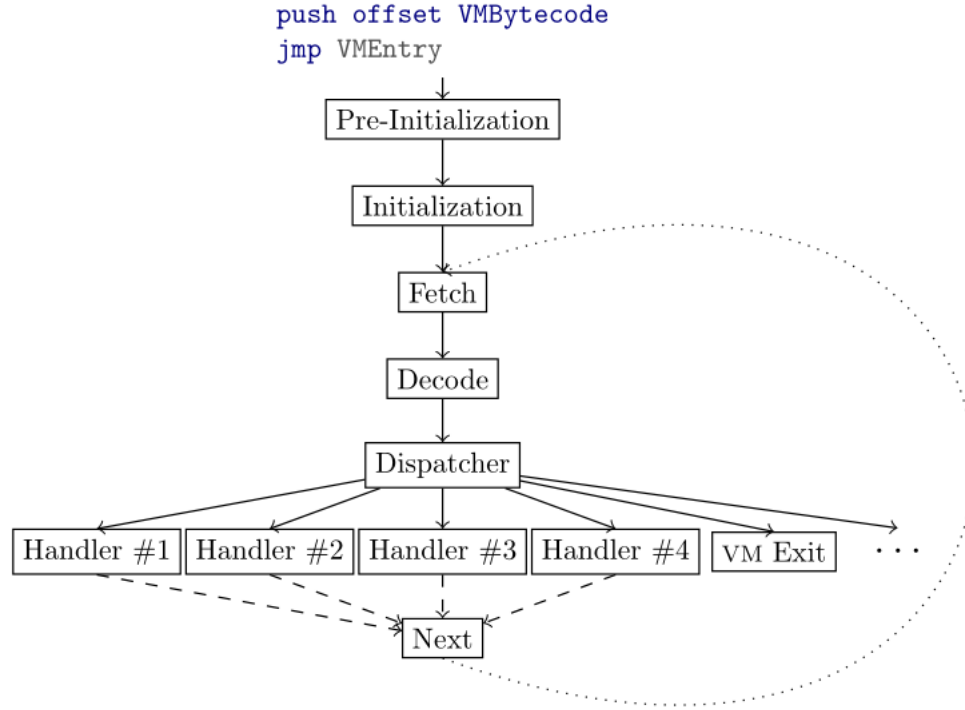


Figure 3: [Taken from here](#)

Since we will be dealing only with FDE part and initialization, I will describe only them

- Initialization - At this step, VM puts all predefined values to their according places in memory and passes important data to an instance such as virtual instruction pointer and default values for registers.
- Fetch - Read instruction pointed to by VIP
- Decoder & Dispatcher - A lot of instructions have different variations such as move data from r2 to r1 or move it from VIP+offset to r1, so we handle such kind of routine here.
- Handler - actual implementation of virtual machine instruction
- Next - step to fetch stage until VM Exit handler

For more precise information, I strongly encourage you to follow the link in the caption.

2.1 Kata VM architecture

Now that we know how the majority of virtual machines work, let's investigate our subject.

```
1  call    fcn.000012d0 ; invoke virtual machine
2  test    al, al
3  jne     0x11ad ; jump to fail message
4  lea     rdi, [0x00006b22] ; pointer to "you won" message
5  call    fcn.000010c0 ; print message
6  jmp     0x1195 ; jump to exit
7  jne     0x11ad ; jump to fail message
8  lea     rdi, [0x00006b22] ; pointer to "you failed" message
9  call    fcn.000010c0 ; print message
10 jmp     0x1195 ; jump to exit
```

Listing 1: main function

The code is pretty straightforward - we call virtual machine if it returns 0 we have entered the right value, otherwise failed. Now let's investigate the virtual machine itself:

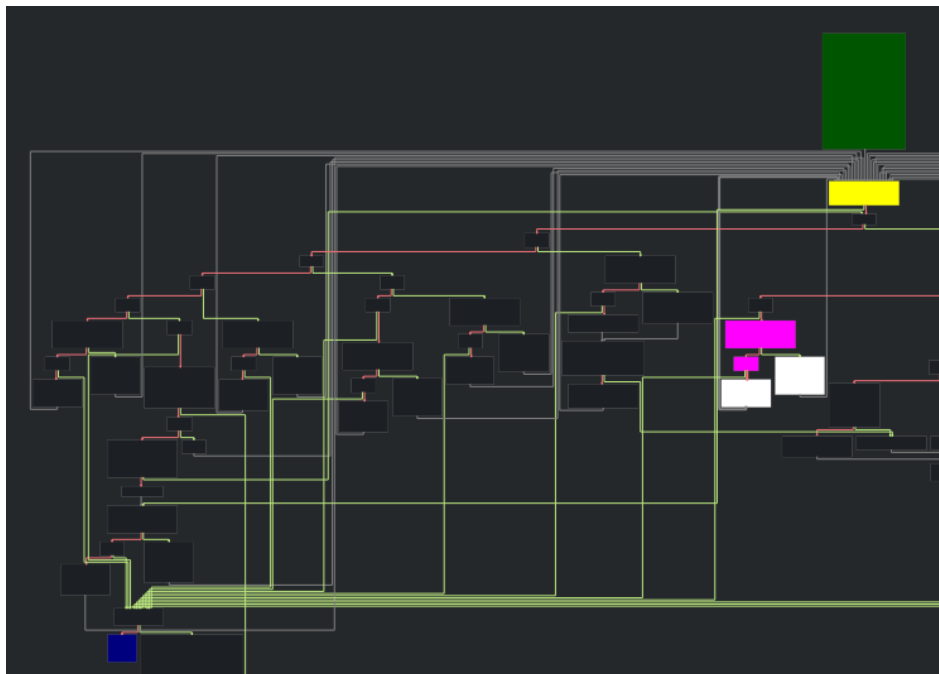


Figure 4: Virtual machine overview

- Green - VM_ENTRY function. It allocates space on stack, copies byte-code and sets default values to registers

- Yellow - get instruction from bytecode pointed to by virtual instruction pointer - it is `rdx` in our case.
- Pink - parse instruction bytecode and redirect control flow to appropriate instruction realization
- White - realizations of instruction. Here most instructions' realizations separated either into "operate on data taken from bytecode and store result in register" or "operate on data from register and store in another register"
- Blue - `VM_EXIT` function which destructs virtual machine's stack and returns success/fail state based on input.

2.2 Investigating where does function result come from

Let's take a look at how verification process looks like - first, consider the `VM_EXIT` listing:

```

1 add rsp, 0x3b60
2 mov eax, r12d
3 pop rbx
4 pop rbp
5 pop r12
6 ret

```

Listing 2: `VM_EXIT`

As you can see the value to `eax` comes from `r12d`. Let's take a look at places where this register is modified:

```

1 0x0000130f xor r12d, r12d ; set initial value to 0 at VM_ENTRY
2 0x00001b8f mov r12d, 1 ; set 1 if call to fcn.00001100 (read system
   call) is not equal to some value in stack/[VIP+2]
3 0x00001589 cmovne r12d, eax ; eax is 1. Note that this instruction
   compares something in stack
4 0x0000152e cmovne r12d, eax ; some coparison instruction that is
   sibling of 0x1589

```

Listing 3: read/write locations

Ahead are all places that somehow interact with `r12d` in this function, so I suggest taking a closer look at those functions:

```

1 xor edi, edi ; file descriptor
2 mov rdx, rbp ; number of symbols to read
3 lea rsi, [rbx + rax*4] ; destination - rbx is top of virtual stack
   and it equals to rsp
4 call fcn.00001100 ; read()
5 cmp rbp, rax ; is number of written symbols equal to expected? If
   not - set r12d = 1

```

Listing 4: `VREAD()` procedure

```

1 mov edi, dword [rdx + 2] ; take some dword from VM bytecode
2 cmp dword [rsp + rax*4], edi ; compare it to some value in stack
3 mov eax, 1
4 cmovne r12d, eax ; set r12d = 1 if they are not equal

```

Listing 5: compare procedure

The other compare function is almost equal except that it takes two values in the stack. Now it's time to run the debugger and learn how many symbols our virtual machine takes:

```

- offset -      0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffd0e09ceb0 3132 3334 0000 0000 0400 0000 0000 0000 1234.....
0x7ffd0e09cec0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffd0e09ced0 9dd5 090e fd7f 0000 0000 0000 0000 0000 .....
0x7ffd0e09cee0 f6ce 49d3 ace7 aa06 29d3 ace7 fa85 ef15 ..I.....).....
0x7ffd0e09cef0 0124 f6c2 d964 29b8 4aca 479b d647 30ef .$ ... d).J.G..G0.
0x7ffd0e09cf00 3901 4ff6 06cb 92bc 2f8b c2eb 92bc 2fef 9.0...../...../

```

Figure 5: stdin read routine

Immediately after that, our virtual machine invokes another VREAD instruction, so in the end, it consumes 8 bytes. It is not quite right because much further in VM bytecode it reads another 4+4 bytes, so probably it was much better to use strace to catch all read() calls, but let's go further

3 Solving using automated tools

I knew this is a relatively simple virtual machine, so I've decided to give a chance to automated analyzers

3.1 Symbolic execution

At this moment I set up angr instance and pointed it to find input which will lead execution to "You won" branch, but it consumed 60 GB of space in 8 hours and I had to kill it, so in my situation this tool was not really useful.

Note that efficiency of symbolic execution tools highly depends on the way you write scripts for them, so it is likely that it was my fault. Also symbolic execution technique suffers from path explosion problem, so choosing this tool was not really wise in this situation.

```

0[|] 1.9%
1[|] 0.0%
2[|] 99.3%
3[|] 2.0%
Mem[|] 4.01G/4.69G
Swp[|] 53.1G/58.8G

```

3.2 Fuzzing

At this point, I already knew there were lots of options among fuzzing engines. My idea was not to pass data to stdin, but rather write data directly to memory directly after `read()` and causing the crash. For this purpose snapshot-based engines fit well, but the problem is, my processor is quite old, so it doesn't support intel PT technology, so I have to stick to more conventional solutions - in my case it was `AFL++`.

I didn't know that if there was passed overhead data to `read()`, it won't be deleted by OS, but rather passed to next `read()` call, so I patched the binary code, and here I will explain how most instructions of bytecode look like.

Let's assume we have this `VMOVDWORD` instruction in bytecode which is true because I took it from our program:

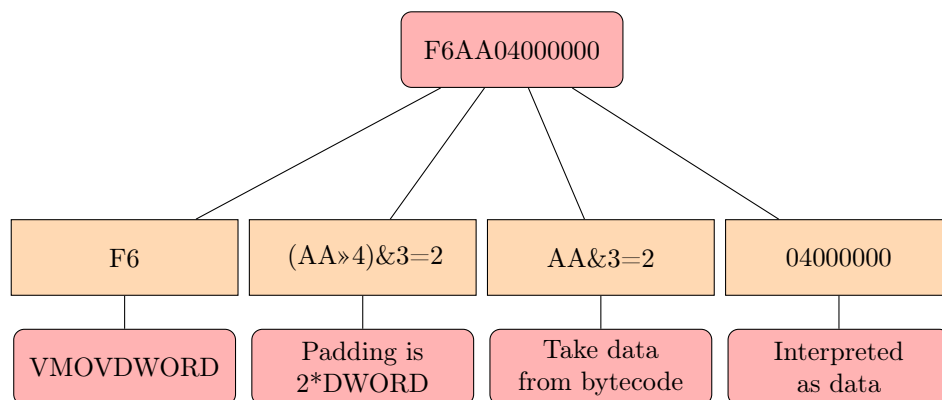


Figure 6: command parsing overview

- `VMOVDWORD` - is a bytecode mnemonic that indicates we are dealing with move operation
- `Padding` - tells the virtual processor how much to step from stack top in multiples of `DWORD` - smallest virtual machine register size.
- `Branch` - tells from which source to take data - it could be either bytecode or virtual register identified by another padding value
- `Bytecode data/source virtual register` - block from which VM takes data.

```

00003690: 15 EF 39 01 F5 F6 86 93 E2 AB C7 AA C6 55 E2 AB ..9.....U..
000036A0: C7 C9 B1 EF 2D 01 3D F6 02 23 8F 81 D4 AA 8A 03 ...-.=.#....
000036B0: 8F 81 D4 79 CF EF 35 01 72 F6 AA 08 00 00 00 5D ...y.5.r.[...]
000036C0: 8C 8C 5D 94 A9 F6 60 60 93 44 F6 BC D3 BC FB 7C ...]...`D....|
000036D0: A6 04 8F 0B C7 04 B1 32 05 D2 34 AC 4A 26 21 6E .....2..4.J&ln
000036E0: B8 80 3F 4A 32 5D 29 68 A2 73 8B E0 F6 D5 EB 07 ...?J2])h.s....
000036F0: 16 A5 2B 6E F1 60 F6 30 D6 07 13 4A 72 B1 FB 9F ...+n..0...lr...

```

Figure 7: Patching bytecode. (Also patched instruction length to avoid second read())

Of course, there are more simple instructions as well as more complex, but this example is enough to get a general idea of how virtual machine understands provided bytecode. After finding and patching the bytecode instruction, I ran a fuzzing session that lasted 3+ days and found nothing except a bunch of hangs which was not quite expected, so I had to understand the reasons for such behavior - to accomplish that, I had to reverse engineer bytecode.

4 Semi-automated approach

The process was quite repetitive for this VM, so I will omit it. If you are interested in my disassembler, you can find it in the `disassembler.py` file in my GitHub repo as well as other notes made in [Obsidian](#).

4.1 Disassembler

I will attach here only the most important things because the whole disassembly is quite extensive.

```

1 0x248: (0xf6); VMOVDWORD r0, 588481e1 // Move dword1
2 0x24e: (0xaa); VSUBDWORD r0, 388481e1 // Move dword2
3 0x256: (0xef); VWRITE dst:1, src:r0, size:1 // Call write() - this
   will print ' '
4 0x25a: (0xf6); VMOVDWORD r0, 4502064c
5 0x260: (0x8b); VXOR r0, 6502064c
6 0x266: (0xef); VWRITE dst:1, src:r0, size:1

```

Listing 6: How writes to stdout are handled

Regarding hangs - I almost immediately found the source of the problem. It was that actually there are 4 calls to `read()`, where each syscall takes 4 bytes as input, so I slightly adjusted my fuzzing settings - and kept reversing the bytecode.

```

1 0x2111: (0xf6); VMOVDWORD r2, 04000000
2 0x2114: (0x5d); VREAD src: 0, dst:r1, size:r2
3 0x2117: (0x5d); VREAD src: 0, dst:r0, size:r2
4 0x211d: (0xf6); VMOVDWORD r2, r1
5 0x2122: (0xf6); VMOVDWORD r3, r2

```

Listing 7: How reads from stdin are handled

The hardest part was understanding what happens after read syscall, because the information mutation pattern was unknown to me. After few days of fruitless attempts to understand what is happening in assembly code, I guessed that probably this is not a self-written algorithm, so finally, I googled some eye-catching details that this algorithm has like shl « 4 and shr » 5 and immediately found the implemented algorithm - It was either TEA or some derivative like XTEA/XXTEA. The wiki even provided us with the decryption script which was quite simple:

```

1 void decrypt (uint32_t v[2], const uint32_t k[4]) {
2     uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i; /* set up; sum
3     is (delta << 5) & 0xFFFFFFFF */
4     uint32_t delta=0x9E3779B9; /* a key
5     schedule constant */
6     uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3]; /* cache key */
7     for (i=0; i<32; i++) { /* basic cycle
8     start */
9         v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
10        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
11        sum -= delta; /* end cycle */
12    }
13    v[0]=v0; v[1]=v1;
14 }

```

Listing 8: TEA decryption algorithm

I had to find 7 important things in this code - 4 dword keys, delta value and 2 dwords of encrypted text. It was quite easy with disassembled code - they are present in the listing:

```

1 ...
2 0x3a61: (0x4a); VADD r2, 216eb880 // key 1
3 0x3a68: (0x4a); VADD r3, 5d2968a2 // key 2
4 ...
5 0x3abd: (0x4a); VADD r0, 2df271f1 // key 3
6 0x3ac4: (0x4a); VADD r1, 943ca128 // key 4
7 ...
8 0x3afa: (0x4a); VADD r2, b1fb9fe0 // delta
9 ...
10 0x3b0a: (0xc3); VCMPLT r2, 2d08e76c r12d = 1 if not equal //
    encrypted DWORD 1.1
11 0x3b11: (0xc3); VCMPLT r3, 2f493ac0 r12d = 1 if not equal //
    encrypted DWORD 1.2
12 ...
13 0x2103: (0xc3); VCMPLT r2, 337698fb r12d = 1 if not equal //
    encrypted DWORD 2.1
14 0x210a: (0xc3); VCMPLT r3, 880450a2 r12d = 1 if not equal //
    encrypted DWORD 2.2
15 ...

```

Listing 9: Key constants

Important note - we need to pass only two dwords per launch for decryption routine because actually in bytecode encrypting operation separated into 2 stages - encrypt 8 bytes, then check that equality evaluates to 0, if so - encrypt another

8 bytes, otherwise set r12d to 1. After inserting gathered data I launched the decryption sequence and got totally wrong answer which was quite upsetting. After a long time of reading VM disassembled code, I noticed that actually at some stage value in the register is shifted by 3 instead of 5, the same situation with another register. Fixing the algorithm is quite easy, the most important part here is to precisely count the loop stage and subtract this value from 32, because during the decryption process we are stepping in reverse order.

```

1  #include <stdint.h>
2  #include <stdio.h>
3
4  void decrypt(uint32_t v[2], const uint32_t k[4]) {
5      uint32_t v0=v[0], v1=v[1], i; /* set up; sum is (delta << 5)
6      & 0xFFFFFFFF */
7      uint32_t sum=0x1c13ff7620;
8      uint32_t delta=0xe09ffbb1; /* a key
9      schedule constant */
10     uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3]; /* cache key */
11     for (i=0; i<32; i++) { /* basic cycle
12     start */
13         if ( i == 9){
14             v1 -= ((v0<<2) + k2) ^ (v0 + sum) ^ ((v0>>3) + k3);
15         }
16         else{
17             v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
18         }
19         v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
20         sum -= delta;
21     } /* end cycle */
22     v[0]=v0; v[1]=v1;
23     printf("%x, %x\n", v[0], v[1]); // later i passed each printed
24     byte to python chr()
25 }
26
27 int main(int argc, char *argv[])
28 {
29     uint32_t v[2] = {0x6ce7082d, 0xc03a492f};
30     uint32_t k[4] = {0x80b86e21, 0xa268295d, 0xf171f22d, 0x28a13c94
31     };
32     decrypt(v, k);
33     return 0;
34 }

```

Listing 10: Fixed TEA decryption algorithm

```
» echo -n 'xNVa2_N07_t3aAlg' | ./KataVM_L1
| Towel's KataVM: Level 1 |
>>
[+] Correct!
```

Figure 8: Solved challenge

5 As a conclusion

Analyzing virtual machines is very time-consuming, the biggest error in my case was wasting time on patching binary, so it is worth being aware of how your OS manages passed input to programs. Also, after realizing that this VM uses an effective crypto algorithm to preserve data, it became quite obvious that fuzzer and angr weren't useful in this case, so I immediately turned these tools off.

5.1 Links

[Tool for automated binary rewriting](#)
[Neat hex editor](#)
[Detect It Easy](#)
[VM analysis and bytecode disassembly article](#)
[My GitHub page](#)
[Angr symbolic execution framework](#)
[Another solution from crackmes.one \(password:crackmes.one\)](#)
[Other materials dedicated to this research](#)
[TEA encryption algorithm](#)