



• [Baleful] •

. Introduction .

Solve

Hint

Review

This program seems to be rather delightfully twisted! Can you get it to accept a password? We need it to get access to some Daedalus Corp files.

. Task description .

This is the description about the task. The hint indicates something about Packed program and to use the packer to get the unpacked code.

It seems to be an easy task to complete but...

. First Contact .

Start using a hex editor. And inspect the contest of the binary. Fire *radare2* (from now *r2*) and type “px 400” to show a hex dump of file:

```
root@vmkalix86 ~/picoctf# r2 baleful
Warning: Cannot initialize section headers
Warning: Cannot initialize strings table
-- Hold on, this should never happen!
[0x00c02038]> px 400
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00c02038 e80f 0200 0060 8b74 2424 8b7c 242c 83cd .....`t$$$.|$,..
0x00c02048 ffeb 0f90 9090 9090 8a06 4688 0747 01db .....F..G..
0x00c02058 7507 8b1e 83ee fc11 db8a 0772 ebb8 0100 u.....r....
0x00c02068 0000 01db 7507 8b1e 83ee fc11 db11 c001 .....u.....
0x00c02078 db73 ef75 098b 1e83 eefc 11db 73e4 31c9 .s.u.....s.l.
0x00c02088 83e8 0372 0dc1 e008 8a06 4683 f0ff 7476 ...r.....F...tv
0x00c02098 89c5 01db 7507 8b1e 83ee fc11 db11 c901 .....u.....
0x00c020a8 db75 078b 1e83 eefc 11db 11c9 7520 4101 .u.....u A.
0x00c020b8 db75 078b 1e83 eefc 11db 11c9 01db 73ef .u.....s.
0x00c020c8 7509 8b1e 83ee fc11 db73 e483 c102 81fd u.....s.....
0x00c020d8 00f3 ffff 83d1 018d 142f 83fd fc8a 040f ...../.....
0x00c020e8 760e 8a02 4288 0747 4975 f7e9 5eff ffff v...B..GIu..^...
0x00c020f8 8b02 83c2 0489 0783 c704 83e9 0477 f101 .....w...
0x00c02108 cfe9 48ff ffff 8b54 2424 0354 2428 39d6 ..H....T$$$.T$(9.
0x00c02118 7401 482b 7c24 2c8b 5424 3089 3a89 4424 t.H+|$,.T$0...D$
0x00c02128 1c61 c361 c30a 0024 496e 666f 3a20 5468 .a.a...$Info: Th
0x00c02138 6973 2066 696c 6520 6973 2070 6163 6b65 is file is packe
0x00c02148 6420 7769 7468 2074 6865 2055 5058 2065 d with the UPX e
0x00c02158 7865 6375 7461 626c 6520 7061 636b 6572 xecutable packer
0x00c02168 2068 7474 703a 2f2f 7570 782e 7366 2e6e http://upx.sf.n
0x00c02178 6574 2024 0a00 2449 643a 2055 5058 2033 et $..$Id: UPX 3
0x00c02188 2e39 3120 436f 7079 7269 6768 7420 2843 .91 Copyright (C
0x00c02198 2920 3139 3936 2d32 3031 3320 7468 6520 ) 1996-2013 the
0x00c021a8 5550 5820 5465 616d 2e20 416c 6c20 5269 UPX Team. All Ri
0x00c021b8 6768 7473 2052 6573 6572 7665 642e 2024 ghts Reserved. $
[0x00c02038]>
```

We can see how the program was compressed with *UPX*. Using the latest version of *UPX*, and the `-d` command retrieve the unpacked executable, it is trivial.

. Analysis .

Now let's inspect the unpacked executable.

Load the unpacked file into *r2* and type “aa” to do the auto analysis, “s entry0” to set the cursor (block) at entry address and “pd 14@entry0” to disassemble 14 instructions from “entry0”.

```
root@vmkalix86:~/picocf/dbgsrv# r2 baleful
-- Thank you for using radare2. Have a nice night!
[0x08048540]> aa
[0x08048540]> s entry0
[0x08048540]> pd 14@entry0
; [12] va=0x08048540 pa=0x00000540 sz=6236 vsz=6236 rwx=-r-x .text
/ (fcn) entry0
section..text:
0x08048540 3led      xor ebp ebp
0x08048542 5e        pop esi
0x08048543 89e1      mov ecx esp
0x08048545 83e4f0    and esp 0xffffffff
0x08048548 50        push eax
0x08048549 54        push esp
0x0804854a 52        push edx
0x0804854b 68609d0408 push 0x8049d60
0x08048550 68f09c0408 push fcn.08049cf0
0x08048555 51        push ecx
0x08048556 56        push esi
0x08048557 68829c0408 push main ; "U..WS....." @ 0x8049c82
0x0804855c e88fffff  call sym.imp.__libc_start_main
\
0x08048561 f4        hlt
[0x08048540]>
```

We could have loaded directly into *r2* and do a “px 14”, but in this example we've used three commands of *r2*:

- “aa”: Auto analysis, try to detect functions, imports, exports, strings.
- “s”: Seek command tells to *r2* to set the address of current block.
- “pd”: Perform a disassemble, if you use this command without parameters it does a disassemble of the current block. The block always is showed at prompt “[0x08048540]>”. You can tell to *pd* to disassemble *N* instructions appending the numbers of instructions to show: “pd 10”. And you can instruct to *pd* to show a disassemble of a specific address, without change the actual block, just using “pd @0x401000”. And lastly you can tell to show *N* instructions from a specific address “pd 20@401000”.

Doing a “pd 14@entry0” r2 shows a disassemble of start function:

```

/ (fcn) entry0 34
|      ;-- section..text:
|      0x08048540  31ed      xor ebp, ebp
|      0x08048542  5e        pop esi
|      0x08048543  89e1      mov ecx, esp
|      0x08048545  83e4f0    and esp, 0xffffffff
|      0x08048548  50        push eax
|      0x08048549  54        push esp
|      0x0804854a  52        push edx
|      0x0804854b  68609d0408 push 0x8049d60
|      0x08048550  68f09c0408 push fcn.08049cf0
|      0x08048555  51        push ecx
|      0x08048556  56        push esi
|      0x08048557  68829c0408 push main      ; "U..WS....." @ 0x8049c82
|      0x0804855c  e88fffff  call sym.imp.__libc_start_main
|      sym.imp.__libc_start_main(unk, unk, unk, unk, unk, unk, unk, unk)
\      0x08048561  f4        hlt

```

Here we can see a call to “sym.imp.__libc_start_main”. Then the main function must be at 0x8049C82 (take in mind the parameters passed to function, first parameter must point to main function code).

Now disassemble the address parsed as first parameter 0x8049c82 to inspect the code: “pd 29@0x8049c82” or “pd 29@main”:

```

;-- main:
0x08049c82  55        push ebp
0x08049c83  89e5      mov ebp, esp
0x08049c85  57        push edi
0x08049c86  53        push ebx
0x08049c87  83e4f0    and esp, 0xffffffff
0x08049c8a  81ec90000000 sub esp, 0x90
0x08049c90  65a114000000 mov eax, dword gs:[0x14]
0x08049c96  8984248c000. mov dword [esp + 0x8c], eax
0x08049c9d  31c0      xor eax, eax
0x08049c9f  8d442410  lea eax, dword [esp + 0x10]
0x08049ca3  89c3      mov ebx, eax
0x08049ca5  b800000000 mov eax, 0
0x08049caa  ba1f000000 mov edx, 0x1f
0x08049caf  89df      mov edi, ebx
0x08049cb1  89d1      mov ecx, edx
0x08049cb3  f3ab      rep stosd dword es:[edi],
0x08049cb5  8d442410  lea eax, dword [esp + 0x10]
0x08049cb9  890424    mov dword [esp], eax
0x08049cbc  e8caecffff call 0x804898b
0x080498b(unk, unk, unk) ; entry0
0x08049cc1  b800000000 mov eax, 0
0x08049cc6  8b94248c000. mov edx, dword [esp + 0x8c]
0x08049ccd  65331514000. xor edx, dword gs:[0x14]
,=< 0x08049cd4  7405      je 0x8049cdb
| 0x08049cd6  e8e5e7ffff call sym.imp.__stack_chk_fail
| 0x080484c0() ; sym.imp.__stack_chk_fail
`-> 0x08049cdb  8d65f8    lea esp, dword [ebp - 8]
0x08049cde  5b        pop ebx
0x08049cdf  5f        pop edi
0x08049ce0  5d        pop ebp
0x08049ce1  c3        ret

```

This main function calls to another function at 0x8049CBC, “pd 1@0x8049cbc”:

```
0x08049cbc  e8caecffff  call 0x804898b
```

And the function at 0x804898B seems to be the real “challenge”, let's go to see what happen inside this one, “pd 38@0x804898b”:

```
|          ; CALL XREF from 0x08049cbc (unk)
/ (fcn) fcn.0804898b 4855
|          0x0804898b  55          push ebp
|          0x0804898c  89e5        mov ebp, esp
|          0x0804898e  81ecc8000000 sub esp, 0xc8
|          0x08048994  c745cc00100. mov dword [ebp - 0x34], 0x1000 ;
| [0x1000:4]=0xc080b60f
|          0x0804899b  837d0800    cmp dword [ebp + 8], 0          ;
| [0x8:4]=0
|          ,=< 0x0804899f  742a        je 0x80489cb
|          | 0x080489a1  c745d000000. mov dword [ebp - 0x30], 0
|          ,==< 0x080489a8  eb19        jmp 0x80489c3                  ; (fcn.0804898b)
|          ; JMP XREF from 0x080489c7 (fcn.0804898b)
|          .----> 0x080489aa  8b45d0      mov eax, dword [ebp - 0x30]
|          ||| 0x080489ad  cle002      shl eax, 2
|          ||| 0x080489b0  034508      add eax, dword [ebp + 8]
|          ||| 0x080489b3  8b10        mov edx, dword [eax]
|          ||| 0x080489b5  8b45d0      mov eax, dword [ebp - 0x30]
|          ||| 0x080489b8  8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
|          ||| 0x080489bf  8345d001    add dword [ebp - 0x30], 1
|          || ; JMP XREF from 0x080489a8 (fcn.0804898b)
|          |'---> 0x080489c3  837dd01e    cmp dword [ebp - 0x30], 0x1e
|          |====< 0x080489c7  7ee1        jle 0x80489aa
|          ,====< 0x080489c9  eb21        jmp 0x80489ec                  ; (fcn.0804898b)
|          | | ; JMP XREF from 0x0804899f (fcn.0804898b)
|          | |'---> 0x080489cb  c745d400000. mov dword [ebp - 0x2c], 0
|          ,=====< 0x080489d2  eb12        jmp 0x80489e6                  ; (fcn.0804898b)
|          ; JMP XREF from 0x080489ea (fcn.0804898b)
|          .-----> 0x080489d4  8b45d4      mov eax, dword [ebp - 0x2c]
|          ||| 0x080489d7  c784854cfff. mov dword [ebp + eax*4 - 0xb4], 0
|          ||| 0x080489e2  8345d401    add dword [ebp - 0x2c], 1
|          || ; JMP XREF from 0x080489d2 (fcn.0804898b)
|          |'-----> 0x080489e6  837dd41e    cmp dword [ebp - 0x2c], 0x1e
|          |=====< 0x080489ea  7ee8        jle 0x80489d4
|          | | ; JMP XREF from 0x080489c9 (fcn.0804898b)
|          |'-----> 0x080489ec  c745c800f00. mov dword [ebp - 0x38], 0xf000
|          |          0x080489f3  c745d800000. mov dword [ebp - 0x28], 0
|          |          0x080489fa  c745ec00000. mov dword [ebp - 0x14], 0
|          |          0x08048a01  c745f000000. mov dword [ebp - 0x10], 0
|          |          0x08048a08  c745f400000. mov dword [ebp - 0xc], 0
|          |          0x08048a0f  c745e800000. mov dword [ebp - 0x18], 0
|          |          0x08048a16  8b45e8      mov eax, dword [ebp - 0x18]
|          |          0x08048a19  8945e4      mov dword [ebp - 0x1c], eax
|          |          0x08048a1c  8b45e4      mov eax, dword [ebp - 0x1c]
|          |          0x08048a1f  8945e0      mov dword [ebp - 0x20], eax
|          |          0x08048a22  8b45e0      mov eax, dword [ebp - 0x20]
|          |          0x08048a25  8945dc      mov dword [ebp - 0x24], eax
|          ,=====< 0x08048a28  e93a120000  jmp 0x8049c67                  ; (fcn.0804898b)
|          | ; JMP XREF from 0x08049c74 (fcn.0804898b)
|          |
```

The first instructions of this function are initializing stack variables, they can be initialized with 0 or with the values given at arg0 as pointer.

This code is a "for from 0 to 0x1E", and it jumps to 0x80489EC, where still it's doing more initializations and finally jumps to 0x8049c67.

We do a disassemble of this offset: "pd 8@0x8049c67":

```
| ; JMP XREF from 0x08048a28 (fcn.0804898b)
| 0x08049c67 8b45cc mov eax, dword [ebp - 0x34]
| 0x08049c6a 05c0c00408 add eax, 0x804c0c0
| 0x08049c6f 0fb600 movzx eax, byte [eax]
| 0x08049c72 3c1d cmp al, 0x1d
|=< 0x08049c74 0f85b3edffff jne 0x8048a2d
| 0x08049c7a 8b854cffffff mov eax, dword [ebp - 0xb4]
| 0x08049c80 c9 leave
| 0x08049c81 c3 ret
```

This chunk of code is loading a value from an address pointed by [ebp-0x34]+0x804c0c0, and it's compared with value 0x1d. If they don't match then jump to another address, else the function ends. This is a check for exit from VM function.

Let's examine the jne landing address "pd 8@0x8048a2d":

```
| ; JMP XREF from 0x08049c74 (fcn.0804898b)
| 0x08048a2d 8b45cc mov eax, dword [ebp - 0x34]
| 0x08048a30 05c0c00408 add eax, 0x804c0c0
| 0x08048a35 0fb600 movzx eax, byte [eax]
| 0x08048a38 0fbec0 movsx eax, al
| 0x08048a3b 83f820 cmp eax, 0x20
|=< 0x08048a3e 0f871e120000 ja 0x8049c62
| 0x08048a44 8b0485d49d0. mov eax, dword [eax*4 + 0x8049dd4]
| 0x08048a4b ffe0 jmp eax
```

It's clear now, here is a big switch case.

We currently have all this information:

- An initialization of 0x1E values to 0 or a specified value.
- A check to determine the function to exit
- A big Switch, testing the extracted byte to do a jump to specific code.

And if you still keep analyzing each case of the table at address 0x8049dd4 you can see how they return to 0x8049c67, at the end of each one, to select another byte and determine the new "instruction/op-code" to execute.

This is, without doubt, the behavior of a Virtual Machine.

. Virtual Machine Analysis .

. What is a Virtual Machine .

“A **virtual machine** is a software computer that, like a physical computer, runs an operating system and applications.”

In our case a virtual machine is a software code that, like a physical processor can run a determinate instruction set.

Every processor needs some basic components:

- *Instruction set*
- *Registers*
- *Memory*
- *I/O devices*

To continue with the analysis will be necessary to locate all these components.

. General Analysis .

At this point we can guess that routine at address “0x0804898B” is without doubt the VM code interpreter.

Now if you remember about the initial analysis, we've found the next code at address 0x8049C6 (where a byte is loaded and compared with the value 0x1d) .

“pd 8@0x809c67”:

```
| ; JMP XREF from 0x08048a28 (fcn.0804898b)
| 0x08049c67 8b45cc mov eax, dword [ebp - 0x34]
| 0x08049c6a 05c0c00408 add eax, 0x804c0c0
| 0x08049c6f 0fb600 movzx eax, byte [eax]
| 0x08049c72 3c1d cmp al, 0x1d
|=< 0x08049c74 0f85b3edffff jne 0x8048a2d
| 0x08049c7a 8b854cffffff mov eax, dword [ebp - 0xb4]
| 0x08049c80 c9 leave
| 0x08049c81 c3 ret
```

The code above loads a byte pointed by offset 0x804c0c0 + [ebp+var_34] then it's compared with 0x1d and jumps to 0x8048a2d if the cmp fails.

“pd 8@0x8048a2d”:

```
| ; JMP XREF from 0x08049c74 (fcn.0804898b)
| 0x08048a2d 8b45cc mov eax, dword [ebp - 0x34]
| 0x08048a30 05c0c00408 add eax, 0x804c0c0
| 0x08048a35 0fb600 movzx eax, byte [eax]
| 0x08048a38 0fbec0 movsx eax, al
| 0x08048a3b 83f820 cmp eax, 0x20
|=< 0x08048a3e 0f871e120000 ja 0x8049c62
| 0x08048a44 8b0485d49d0. mov eax, dword [eax*4 + 0x8049dd4]
| 0x08048a4b ffe0 jmp eax
```

The code is same as previous but the `cmp` is made with `0x20`, and if the value is lower then jumps to the address at table `dword [eax*4 + 0x8049dd4]`.

It's clear, the first chunk of code tests for a byte `0x1d`, this value tells the VM to end. And in second chunk, the `cmp` was made with `0x20` that is the maximum number of *op-codes*, and if the value is lower, jumps to the code to manage this instruction of the VM (*op-code*).

Well, from this little analysis we have:

- Total number of instructions: `0x20` instructions
- Value of the instruction to end the VM code execution: `0x1d`
- Address of all instructions: table at `0x8049dd4`
- *Instruction pointer register*: `[ebp-var_34]`
- Virtual machine base: `0x804C0C0`

This is the code from the beginning of the main VM routine (`0x804898b`):

```

/ (fcn) fcn.0804898b 4855
|      0x0804898b 55      push ebp
|      0x0804898c 89e5      mov ebp, esp
|      0x0804898e 81ecc8000000 sub esp, 0xc8
|      0x08048994 c745cc00100. mov dword [ebp - 0x34], 0x1000 ;
|      0x0804899b 837d0800      cmp dword [ebp + 8], 0 ;
|      ,=< 0x0804899f 742a      je 0x80489cb
|      | 0x080489a1 c745d000000. mov dword [ebp - 0x30], 0
|      ,==< 0x080489a8 eb19      jmp 0x80489c3 ; (fcn.0804898b)
|      ; JMP XREF from 0x080489c7 (fcn.0804898b)
|      .----> 0x080489aa 8b45d0      mov eax, dword [ebp - 0x30]
|      ||| 0x080489ad c1e002      shl eax, 2
|      ||| 0x080489b0 034508      add eax, dword [ebp + 8]
|      ||| 0x080489b3 8b10      mov edx, dword [eax]
|      ||| 0x080489b5 8b45d0      mov eax, dword [ebp - 0x30]
|      ||| 0x080489b8 8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
|      ||| 0x080489bf 8345d001      add dword [ebp - 0x30], 1
|      || ; JMP XREF from 0x080489a8 (fcn.0804898b)
|      |`--> 0x080489c3 837dd01e      cmp dword [ebp - 0x30], 0x1e
|      |`==< 0x080489c7 7ee1      jle 0x80489aa
|      ,====< 0x080489c9 eb21      jmp 0x80489ec ; (fcn.0804898b)
|      | | ; JMP XREF from 0x0804899f (fcn.0804898b)
|      | |`-> 0x080489cb c745d400000. mov dword [ebp - 0x2c], 0
|      ,===== 0x080489d2 eb12      jmp 0x80489e6 ; (fcn.0804898b)
|      ; JMP XREF from 0x080489ea (fcn.0804898b)
|      .-----> 0x080489d4 8b45d4      mov eax, dword [ebp - 0x2c]
|      ||| 0x080489d7 c784854cfff. mov dword [ebp + eax*4 - 0xb4], 0
|      ||| 0x080489e2 8345d401      add dword [ebp - 0x2c], 1
|      || ; JMP XREF from 0x080489d2 (fcn.0804898b)
|      |`-----> 0x080489e6 837dd41e      cmp dword [ebp - 0x2c], 0x1e
|      |`===== 0x080489ea 7ee8      jle 0x80489d4
|      | | ; JMP XREF from 0x080489c9 (fcn.0804898b)
|      |`-----> 0x080489ec c745c800f00. mov dword [ebp - 0x38], 0xf000

```

Shows how the VM initialize its registers, this initialization can be done by two ways determined by `arg0`:

- If `arg0` is null, all VM registers will be initialized with value 0.
- If `arg0` has a value, it must be a pointer to `0x1e dwords`. This one is used to initialize the registers with values from another VM call or parsing the current processor registers; this way VM can operate with real registers.

Great, now we know the total number of registers and where they are allocated (`0x1e` registers and all of them are allocated into the stack). Always there are three special registers we need:

- *Stack Register*
- *Eflags Register*
- *Instruction Pointer Register*

Only take in mind this: all that information becomes useful to us when we start to analyze the op-codes. At this moment the *Instruction Pointer Register* is known from the previous analysis, and its located at `[ebp-34] + 0x804c0c0`.

The next step must be to analyze an op-code to determine how the parameters are handled. Remember that it's a VM and each instruction of the instruction set is named *op-code*. Each op-code that operates with a register or a value must have a parameter to get the VM registers, immediate value, or both.

To illustrate this concept let me take an example from X86:

```
mov eax,1          reg,imm
mov eax,ebx         reg,reg
mov [eax],1        [reg],imm
mov [eax],ebx       [reg],reg
```

Observe how the addressing of `mov [reg],[reg]` doesn't exist in the x86. But in a VM this addressing can be implemented, no is the case of this VM, but none limit to do it.

SkUaTeR

Above you can see the op-code “`mov`” and the different addressing modes (yes it's a simple example, the real `mov` has many more addressing modes). This concept is same into the VM. If an op-code does a `mov` from a reg to another reg, the encode of this type of `mov` must be distinct from the `mov` of a value into reg. This means a byte is required to encode the instruction and another byte or bytes to encode the parameters. Or can be accomplished with a single byte if you can manage all the addressing into one single byte.

That must be kept in mind at the design phase of the VM and try to optimize as most as possible. The analyzed VM always uses a byte to tell the op-code and, if it's required, another one to tell the addressing and the needed values.

After a complete analysis of this vm, the op-code encoding can be done using a single byte for the op-code and the addressing. But I think this guys don't want make we life more harder ;)

SkUaTeR

. Op-codes and parameters analysis .

It's time to take a view into the *op-code* table at address 0x8049dd4. This address contains pointers to code, using next command shows the table.

“pxw 0x24*4@0x8049dd4”:

```
0x08049dd4 0x08048a4d 0x08048a56 0x08048a8f 0x08048bc4 M...V.....
0x08049de4 0x08048cf9 0x08048e2f 0x08048f91 0x080495f5 .... /.....
0x08049df4 0x08049649 0x080490c6 0x080491fb 0x0804959e I.....
0x08049e04 0x08049330 0x08049467 0x080496d1 0x0804969d 0...g.....
0x08049e14 0x080496ec 0x08049715 0x0804973e 0x08049767 .....>...g...
0x08049e24 0x08049790 0x080497b9 0x080497e2 0x080498f0 .....
0x08049e34 0x08049a02 0x08049a86 0x08049ab9 0x08049aec .....
0x08049e44 0x08049b43 0x08049c62 0x08049b92 0x08049bf8 C...b.....
0x08049e54 0x08049c2e 0x3b031b01 0x000000e0 0x0000001b ...../.....
```

Now using this offsets you can inspect each *op-code* to determine what they are doing. Let's start with the first, “pd 2@0x8048a4d”:

```
0x08048a4d 8345cc01 add dword [ebp - 0x34], 1
,<= 0x08048a51 e911120000 jmp 0x8049c67
```

This is easy one, [ebp-0x34] is the *Instruction pointer* and the code are incrementing it, then it is a “NOP”. And don't use any addressing because affect directly to register at instruction pointer, and this register usually can't be accessed via a normal instruction with register manipulation, but remember this always depends on VM design.

Continue with the next value at table “0x8048a56”, and do a disassemble “pd 15@0x8048a56”:

```
0x08048a56 8b45c8 mov eax, dword [ebp - 0x38]
0x08048a59 05c0c00408 add eax, 0x804c0c0
0x08048a5e 8b00 mov eax, dword [eax]
0x08048a60 8945ec mov dword [ebp - 0x14], eax
0x08048a63 837dec00 cmp dword [ebp - 0x14], 0
```

```

,=< 0x08048a67 750b jne 0x8048a74
| 0x08048a69 8b854cffffff mov eax, dword [ebp - 0xb4]
,==< 0x08048a6f e90c120000 jmp 0x8049c80
|`-> 0x08048a74 8b45c8 mov eax, dword [ebp - 0x38]
| 0x08048a77 83c004 add eax, 4
| 0x08048a7a 8945c8 mov dword [ebp - 0x38], eax
| 0x08048a7d 8b45ec mov eax, dword [ebp - 0x14]
| 0x08048a80 8945cc mov dword [ebp - 0x34], eax
| 0x08048a83 c745d800000. mov dword [ebp - 0x28], 0
,===< 0x08048a8a e9d8110000 jmp 0x8049c67

```

This code retrieves a value from VM base, pointed by [ebp-38] (at this moment we don't know relation between the VM and [ebp-38]).

If the loaded value isn't zero the increments in 4 the value of [ebp-38] and finally sets [ebp-34] to the value extracted. As [ebp-34] is the VM *Instruction Pointer*, then this code is extracting a value from the VM context incrementing the pointer to this value and loading into the *Instruction Pointer*.

Hm, it seems to be a RET instruction, and now we know the functionality of [ebp-38] : this is the VM *Stack Pointer* ;)

Take the next value of the table “0x08048a8f” and continue the analysis, “pd 200@0x8048a8f”

```

|| 0x08048a8f 8b45cc mov eax, dword [ebp - 0x34]
|| 0x08048a92 83c001 add eax, 1
|| 0x08048a95 0fb680c0c00. movzx eax, byte [eax + 0x804c0c0] ;
|| 0x08048a9c 0fbec0 movsx eax, al
|| 0x08048a9f 8945f4 mov dword [ebp - 0xc], eax
|| 0x08048aa2 8b45cc mov eax, dword [ebp - 0x34]
|| 0x08048aa5 83c002 add eax, 2
|| 0x08048aa8 0fb680c0c00. movzx eax, byte [eax + 0x804c0c0] ;
|| 0x08048aaf 0fbec0 movsx eax, al
|| 0x08048ab2 8945dc mov dword [ebp - 0x24], eax
|| 0x08048ab5 8b45f4 mov eax, dword [ebp - 0xc]
|| 0x08048ab8 83f801 cmp eax, 1
,====< 0x08048abb 745e je 0x8048b1b
||| 0x08048abd 83f801 cmp eax, 1
,====< 0x08048ac0 7f09 jg 0x8048acb
|||| 0x08048ac2 85c0 test eax, eax
,====< 0x08048ac4 7418 je 0x8048ade
,====< 0x08048ac6 e9d5000000 jmp 0x8048ba0
||`-----> 0x08048acb 83f802 cmp eax, 2
=====< 0x08048ace 747b je 0x8048b4b
||| ||| 0x08048ad0 83f804 cmp eax, 4
=====< 0x08048ad3 0f84a2000000 je 0x8048b7b
=====< 0x08048ad9 e9c2000000 jmp 0x8048ba0
|`-----> 0x08048ade 8b45cc mov eax, dword [ebp - 0x34]
| ||| 0x08048ae1 83c003 add eax, 3
| ||| 0x08048ae4 0fb680c0c00. movzx eax, byte [eax + 0x804c0c0] ;
| ||| 0x08048aeb 0fbec0 movsx eax, al
| ||| 0x08048aee 8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
| ||| 0x08048af5 8945e0 mov dword [ebp - 0x20], eax
| ||| 0x08048af8 8b45cc mov eax, dword [ebp - 0x34]
| ||| 0x08048afb 83c004 add eax, 4
| ||| 0x08048afe 0fb680c0c00. movzx eax, byte [eax + 0x804c0c0] ;
| ||| 0x08048b05 0fbec0 movsx eax, al
| ||| 0x08048b08 8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
| ||| 0x08048b0f 8945e4 mov dword [ebp - 0x1c], eax

```

```

|   ||   0x08048b12      8345cc05      add dword [ebp - 0x34], 5
=====< 0x08048b16      e985000000      jmp 0x8048ba0
|   \-----> 0x08048b1b      8b45cc      mov eax, dword [ebp - 0x34]
|   ||   0x08048b1e      83c003      add eax, 3
|   ||   0x08048b21      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
|   ||   0x08048b28      0fbec0      movsx eax, al
|   ||   0x08048b2b      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
|   ||   0x08048b32      8945e0      mov dword [ebp - 0x20], eax
|   ||   0x08048b35      8b45cc      mov eax, dword [ebp - 0x34]
|   ||   0x08048b38      83c004      add eax, 4
|   ||   0x08048b3b      05c0c00408 add eax, 0x804c0c0
|   ||   0x08048b40      8b00      mov eax, dword [eax]
|   ||   0x08048b42      8945e4      mov dword [ebp - 0x1c], eax
|   ||   0x08048b45      8345cc08      add dword [ebp - 0x34], 8
=====< 0x08048b49      eb55      jmp 0x8048ba0
|   \-----> 0x08048b4b      8b45cc      mov eax, dword [ebp - 0x34]
|   ||   0x08048b4e      83c003      add eax, 3
|   ||   0x08048b51      05c0c00408 add eax, 0x804c0c0
|   ||   0x08048b56      8b00      mov eax, dword [eax]
|   ||   0x08048b58      8945e0      mov dword [ebp - 0x20], eax
|   ||   0x08048b5b      8b45cc      mov eax, dword [ebp - 0x34]
|   ||   0x08048b5e      83c007      add eax, 7
|   ||   0x08048b61      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
|   ||   0x08048b68      0fbec0      movsx eax, al
|   ||   0x08048b6b      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
|   ||   0x08048b72      8945e4      mov dword [ebp - 0x1c], eax
|   ||   0x08048b75      8345cc08      add dword [ebp - 0x34], 8
=====< 0x08048b79      eb25      jmp 0x8048ba0
|   \-----> 0x08048b7b      8b45cc      mov eax, dword [ebp - 0x34]
|   ||   0x08048b7e      83c003      add eax, 3
|   ||   0x08048b81      05c0c00408 add eax, 0x804c0c0
|   ||   0x08048b86      8b00      mov eax, dword [eax]
|   ||   0x08048b88      8945e0      mov dword [ebp - 0x20], eax
|   ||   0x08048b8b      8b45cc      mov eax, dword [ebp - 0x34]
|   ||   0x08048b8e      83c007      add eax, 7
|   ||   0x08048b91      05c0c00408 add eax, 0x804c0c0
|   ||   0x08048b96      8b00      mov eax, dword [eax]
|   ||   0x08048b98      8945e4      mov dword [ebp - 0x1c], eax
|   ||   0x08048b9b      8345cc0b      add dword [ebp - 0x34], 0xb
|   ||   0x08048b9f      90      nop
|   \-----> 0x08048ba0      8b45e4      mov eax, dword [ebp - 0x1c]
|   ||   0x08048ba3      8b55e0      mov edx, dword [ebp - 0x20]
|   ||   0x08048ba6      01c2      add edx, eax
|   ||   0x08048ba8      8b45dc      mov eax, dword [ebp - 0x24]
|   ||   0x08048bab      8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
|   ||   0x08048bb2      8b45dc      mov eax, dword [ebp - 0x24]
|   ||   0x08048bb5      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
|   ||   0x08048bbc      8945d8      mov dword [ebp - 0x28], eax
=====< 0x08048bbf      e9a3100000 jmp 0x8049c67

```

!!! OUCH WTF !!! keep calm and take this analysis with philosophy, that stuff seems bigger than it is.

Surely this *op-code* manages different addressing modes, this is the reason for the big size of the code. Hands to work to continue I'm going to split code in a few parts:

```

||   0x08048a8f      8b45cc      mov eax, dword [ebp - 0x34]
||   0x08048a92      83c001      add eax, 1
||   0x08048a95      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
||   0x08048a9c      0fbec0      movsx eax, al
||   0x08048a9f      8945f4      mov dword [ebp - 0xc], eax
||   0x08048aa2      8b45cc      mov eax, dword [ebp - 0x34]
||   0x08048aa5      83c002      add eax, 2

```

```

    || 0x08048aa8 0fb680c0c00. movzx eax, byte [eax + 0x804c0c0] ;
    || 0x08048aaf 0fbec0      movsx eax, al
    || 0x08048ab2 8945dc      mov dword [ebp - 0x24], eax
    || 0x08048ab5 8b45f4      mov eax, dword [ebp - 0xc]
    || 0x08048ab8 83f801      cmp eax, 1
    ,====< 0x08048abb 745e        je 0x8048b1b
    ||| 0x08048abd 83f801      cmp eax, 1
    ,====< 0x08048ac0 7f09        jg 0x8048acb
    |||| 0x08048ac2 85c0        test eax, eax
    ,====< 0x08048ac4 7418        je 0x8048ade
    ,====< 0x08048ac6 e9d5000000  jmp 0x8048ba0
    ||`-----> 0x08048acb 83f802      cmp eax, 2
    =====< 0x08048ace 747b        je 0x8048b4b
    || ||| 0x08048ad0 83f804      cmp eax, 4
    =====< 0x08048ad3 0f84a2000000 je 0x8048b7b
    =====< 0x08048ad9 e9c2000000  jmp 0x8048ba0

```

This piece of code discards the op-code byte and loads into AL the next byte of the instruction buffer and saves the value into `ebp-0xc`, then loads the next byte from instruction buffer and saves it into `ebp-0x24`, then compares the first extracted byte to decide what to do.

This behavior may be to determine the addressing mode of this instruction and by the moment is composed by [Byte Op-code Number] [Byte addressing mode] [unknown byte].

Now we know what is the goal of this code. It's used to get the parameters and theirs type. Now let's inspect what it does in each case.

. Addressing mode: REG,REG .

```

; extract from execution buffer third byte
0x08048ade mov eax, dword [ebp - 0x34]      ; load into eax vIP
0x08048ae1 add eax, 3                      ; add 3 to vIP
0x08048ae4 movzx eax, byte [eax + 0x804c0c0] ; load into eax the byte
ptr at vIP
0x08048aeb movsx eax, al

; Load into eax a value from the vm reg context ebp-0xb4 is the base of
VM registers.
0x08048aee mov eax, dword [ebp + eax*4 - 0xb4];
; Save the value extracted from VM Reg, into a temp variable
0x08048af5 mov dword [ebp - 0x20], eax

; extract from execution buffer four byte
0x08048af8 mov eax, dword [ebp - 0x34]      ;
0x08048afb add eax, 4                      ;
0x08048afe movzx eax, byte [eax + 0x804c0c0] ;
0x08048b05 movsx eax, al

; Load into eax a value from the vm reg context ebp-0xb4 is the base of
VM registers.
0x08048b08 mov eax, dword [ebp + eax*4 - 0xb4];
; Save the value extracted from VM Reg, into a temp variable
0x08048b0f mov dword [ebp - 0x1c], eax

```

```

; Increment the value of virtual instruction pointer
0x08048b12 add dword [ebp - 0x34], 5 ; increment the vIP in 5
0x08048b16 jmp 0x8048ba0

```

This code is retrieving from VM Registers the registers dictated by the parameters of the op-code, saved into a temporal stack variables, incrementing the instruction pointer and finally jumping to another place. This place may be where the real “instruction” is done, and all this code is only to retrieve the parameters. They are retrieving the addressing of 2 virtual registers and incrementing the virtual instruction pointer in five. Check this out:

[op-code] [p1] [p2] [p3] [p4]

The code extracts p3 and p4 and use theirs values to get VM registers, and as you can see the size is five bytes, just the numbers of bytes what vIP is incremented. Then for this case the encoding looks like this:

1 1 1 1 1
 [op-code] [addressing] [unknown] [reg] [reg]

. Addressing mode: REG,IMM32 .

```

; extract from execution buffer third byte
0x08048b1b mov eax, dword [ebp - 0x34]
0x08048b1e add eax, 3
0x08048b21 movzx eax, byte [eax + 0x804c0c0] ;
0x08048b28 movsx eax, al

; Load into eax a value from the vm reg context ebp-0xb4 is the base of
VM registers and use as index the third byte of execution buffer.
0x08048b2b mov eax, dword [ebp + eax*4 - 0xb4]
; Save the retrieve value in temporal stack var.
0x08048b32 mov dword [ebp - 0x20], eax

; point eax to four byte of the execution buffer
0x08048b35 mov eax, dword [ebp - 0x34]
0x08048b38 add eax, 4
0x08048b3b add eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b40 mov eax, dword [eax]
; Save the retrieve value in temporal stack var.
0x08048b42 mov dword [ebp - 0x1c], eax

; Increment the value of virtual instruction pointer by 8
0x08048b45 add dword [ebp - 0x34], 8
0x08048b49 jmp 0x8048ba0

```

The code is retrieving a VM register indexed by the value of the third byte at instruction buffer, and an immediate dword value at fourth position at instruction buffer. Finally it increments the vIP in 8:

1	1	1	1	4
[op-code]	[addressing]	[unknown]	[reg]	[DWORD]

. Addressing mode: IMM32,REG .

```
; point eax to third byte of the execution buffer
0x08048b4b  mov  eax, dword [ebp - 0x34]
0x08048b4e  add  eax, 3
0x08048b51  add  eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b56  mov  eax, dword [eax]
; Save the retrieve value in temporal stack var.
0x08048b58  mov  dword [ebp - 0x20], eax

; point eax to seven byte of the execution buffer
0x08048b5b  mov  eax, dword [ebp - 0x34]
0x08048b5e  add  eax, 7
; load into EAX the byte at execution buffer
0x08048b61  movzx eax, byte [eax + 0x804c0c0] ;
0x08048b68  movsx eax, al

; Load into eax a value from the vm reg context ebp-0xb4 is the base of
VM registers and use as index the seven byte of execution buffer.
0x08048b6b  mov  eax, dword [ebp + eax*4 - 0xb4]
0x08048b72  mov  dword [ebp - 0x1c], eax

; Increment the value of virtual instruction pointer by 8
0x08048b75  add  dword [ebp - 0x34], 8
0x08048b79  jmp  0x8048ba0
```

The code is retrieving an immediate dword value in third position at instruction buffer, and one VM register indexed by the value of the seventh byte at instruction buffer. Finally it increments the vIP in 8:

1	1	1	4	1
[op-code]	[addressing]	[unknown]	[dword]	[reg]

. Addressing mode: IMM32,IMM32 .

```
; point eax to third byte of the execution buffer
0x08048b7b  mov  eax, dword [ebp - 0x34]
0x08048b7e  add  eax, 3
0x08048b81  add  eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b86  mov  eax, dword [eax]

; Save the retrieve value in temporal stack var.
0x08048b88  mov  dword [ebp - 0x20], eax

; point eax to seven byte of the execution buffer
0x08048b8b  mov  eax, dword [ebp - 0x34]
0x08048b8e  add  eax, 7
0x08048b91  add  eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b96  mov  eax, dword [eax]

; Save the retrieve value in temporal stack var.
0x08048b98  mov  dword [ebp - 0x1c], eax

; Increment the value of virtual instruction pointer by 8
0x08048b9b  add  dword [ebp - 0x34], 0xb
0x08048b9f  nop
```

The code is retrieving an immediate dword value at third position inside instruction buffer, and another dword value at seventh position at instruction buffer. Finally it increments the vIP in 0xb:

1	1	1	4	4
[op-code]	[addressing]	[unknown]	[DWORD]	[DWORD]

. The Real Operation Code .

```

;Load into eax the saved value3
0x08048ba0  mov  eax, dword [ebp - 0x1c]

;load into edx the save value2
0x08048ba3  mov  edx, dword [ebp - 0x20]

; do the operation ADD
0x08048ba6  add  edx, eax

; Load into eax the save value1
0x08048ba8  mov  eax, dword [ebp - 0x24]

; Write into vm reg at index selected by the previous instruction, the
result of the add operation (edx)
0x08048bab  mov  dword [ebp + eax*4 - 0xb4], edx

0x08048bb2  mov  eax, dword [ebp - 0x24]
0x08048bb5  mov  eax, dword [ebp + eax*4 - 0xb4]
0x08048bbc  mov  dword [ebp - 0x28], eax
; Process next op-code
0x08048bbf  jmp  0x8049c67

```

This last chunk of code is where really the operation is made and saved into the context of the VM. With the saved values it does the real operation and then stores the result into VM register using as index the second parameter of op-code. So op-code representation is:

[op-code] [p1] [p2] [p3] [p4]

[ADD] [addressing] [dest_reg] [x] [y]

x and y depend of the addressing mode. Now we have all needed info to make a table with the distinct addressing modes for this op-code. And surely this is applied to another op-codes.

SkUaTeR

This instruction can be represented as:

- if addressing=0: $\text{dest_reg} = \text{reg}[x] + \text{reg}[y]$ (0)
- if addressing=1: $\text{dest_reg} = \text{reg}[x] + \text{imm32}$ (1)
- if addressing=2: $\text{dest_reg} = \text{imm32} + \text{reg}[y]$ (2)
- if addressing=4: $\text{dest_reg} = \text{imm32} + \text{imm32}$ (4)

Now we know how two single op-codes and one other complex addressing op-code manage their parameters from the execution buffer, and in base their addressing byte we know their length. Let me show it in a table:

Op-code #	Name	Length	0	1	2	4
0	NOP	1				
1	RET	1				
2	ADD	5/8/8/11	r0=r1+r2	r0=r1+i32_2	r0=i32_1+r2	r0=i32_1+i32_2

In this table I'm naming the register as r0,r1,r2 and the immediate values as i32_1,i32_2. The value after rX is the index extracted from the byte Xth at execution buffer after addressing byte. And the value after i32_X is a dword at offset Xth after the addressing byte.

SkUaTeR

If at execution buffer there is a byte “0” this is a “nop” and its length is 1.

If at execution buffer there is a byte “1” this is a “ret” and its length is 1.

If at execution buffer there is a byte “2” this is an “add” and we need to look at next byte to know its length.

To get their addressing modes we need look at next byte:

- If next byte is “0” then the length becomes 5 and we need to extract three more bytes to get r0, r1 and r2 pointers.
- If next byte is “1” then the length becomes 8 and we need to extract two more bytes from the execution buffer, these bytes are the indexes for r0 and r1, and a dword to get the value of i32_2.
- And so on. For the next value look the table ...

Building the table for all the op-codes bring us the power needed to get all the “virtual” code and to end with this challenge.

Then keep analyzing the op-code table to discover remaining *op-codes* but now we have a picture of how the addressing are encoded, and it helps to do a quick analysis of remaining instructions.

Get next value of the table “0x08048bc4” and continue the analysis.

“pd 20000x8048bc4”:

```

0x08048bc4      8b45cc      mov eax, dword [ebp - 0x34]
0x08048bc7      83c001      add eax, 1
0x08048bca      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048bd1      0fbec0      movsx eax, al
0x08048bd4      8945f4      mov dword [ebp - 0xc], eax
0x08048bd7      8b45cc      mov eax, dword [ebp - 0x34]
0x08048bda      83c002      add eax, 2
0x08048bdd      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048be4      0fbec0      movsx eax, al
0x08048be7      8945dc      mov dword [ebp - 0x24], eax
0x08048bea      8b45f4      mov eax, dword [ebp - 0xc]
0x08048bed      83f801      cmp eax, 1
0x08048bf0      745e       je 0x8048c50
0x08048bf2      83f801      cmp eax, 1
0x08048bf5      7f09       jg 0x8048c00
0x08048bf7      85c0       test eax, eax
0x08048bf9      7418       je 0x8048c13
0x08048bfb      e9d5000000 jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c00      83f802      cmp eax, 2
0x08048c03      747b       je 0x8048c80
0x08048c05      83f804      cmp eax, 4
0x08048c08      0f84a2000000 je 0x8048cb0
0x08048c0e      e9c2000000 jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c13      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c16      83c003      add eax, 3
0x08048c19      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c20      0fbec0      movsx eax, al
0x08048c23      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048c2a      8945e0      mov dword [ebp - 0x20], eax
0x08048c2d      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c30      83c004      add eax, 4
0x08048c33      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c3a      0fbec0      movsx eax, al
0x08048c3d      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048c44      8945e4      mov dword [ebp - 0x1c], eax
0x08048c47      8345cc05      add dword [ebp - 0x34], 5
0x08048c4b      e985000000 jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c50      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c53      83c003      add eax, 3
0x08048c56      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c5d      0fbec0      movsx eax, al
0x08048c60      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048c67      8945e0      mov dword [ebp - 0x20], eax
0x08048c6a      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c6d      83c004      add eax, 4
0x08048c70      05c0c00408 add eax, 0x804c0c0
0x08048c75      8b00       mov eax, dword [eax]
0x08048c77      8945e4      mov dword [ebp - 0x1c], eax
0x08048c7a      8345cc08      add dword [ebp - 0x34], 8
0x08048c7e      eb55       jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c80      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c83      83c003      add eax, 3
0x08048c86      05c0c00408 add eax, 0x804c0c0
0x08048c8b      8b00       mov eax, dword [eax]
0x08048c8d      8945e0      mov dword [ebp - 0x20], eax
0x08048c90      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c93      83c007      add eax, 7
0x08048c96      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c9d      0fbec0      movsx eax, al
0x08048ca0      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048ca7      8945e4      mov dword [ebp - 0x1c], eax
0x08048caa      8345cc08      add dword [ebp - 0x34], 8
0x08048cae      eb25       jmp 0x8048cd5 ; (fcn.0804898b)
0x08048cb0      8b45cc      mov eax, dword [ebp - 0x34]
0x08048cb3      83c003      add eax, 3
0x08048cb6      05c0c00408 add eax, 0x804c0c0
0x08048cbb      8b00       mov eax, dword [eax]
0x08048cbd      8945e0      mov dword [ebp - 0x20], eax
0x08048cc0      8b45cc      mov eax, dword [ebp - 0x34]
0x08048cc3      83c007      add eax, 7

```

```

0x08048cc6  05c0c00408  add eax, 0x804c0c0
0x08048ccb  8b00        mov eax, dword [eax]
0x08048ccd  8945e4      mov dword [ebp - 0x1c], eax
0x08048cd0  8345cc0b   add dword [ebp - 0x34], 0xb
0x08048cd4  90         nop
0x08048cd5  8b45e4      mov eax, dword [ebp - 0x1c]
0x08048cd8  8b55e0      mov edx, dword [ebp - 0x20]
0x08048cdb  29c2       sub edx, eax
0x08048cdd  8b45dc      mov eax, dword [ebp - 0x24]
0x08048ce0  8994854cff. mov dword [ebp + eax*4 - 0xb4], edx
0x08048ce7  8b45dc      mov eax, dword [ebp - 0x24]
0x08048cea  8b84854cff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048cf1  8945d8      mov dword [ebp - 0x28], eax

```

This code is also similar to previous analyzed, and it's logic, because VM uses always the same method for addressing, and mostly of the code is to manage this. Focus at this address:

```

0x08048cd4  90         nop
0x08048cd5  8b45e4      mov eax, dword [ebp - 0x1c]
0x08048cd8  8b55e0      mov edx, dword [ebp - 0x20]
0x08048cdb  29c2       sub edx, eax
0x08048cdd  8b45dc      mov eax, dword [ebp - 0x24]
0x08048ce0  8994854cff. mov dword [ebp + eax*4 - 0xb4], edx
0x08048ce7  8b45dc      mov eax, dword [ebp - 0x24]
0x08048cea  8b84854cff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048cf1  8945d8      mov dword [ebp - 0x28], eax
0x08048cf4  e96e0f0000 jmp 0x8049c67 ; (fcn.0804898b)

```

Code at “0x08048cdb sub edx, eax” is where the real operation is made, and reveal this op-code, a “SUB”.

All the op-codes with addressing was managed by same manner except “DIV”. For div they add a fourth register to store the remainder, this is the only exception, but doing the analysis how I've showed previously you can determine this without any problem.

And knowing the op-code name and looking the stack variables immerse into the manage of each op-code you can figure the *Eflags*, *Stack*, and other specials registers used by the VM.

. Conditional Jumps & Eflags register .

Lastly I'm going to explain how the conditional jumps works and how this type of op-code reveals the *Eflags* register. “pd 12@0x080496ec”

```

; Load vIP and increment by 1
0x080496ec  8b45cc      mov eax, dword [ebp - 0x34]
0x080496ef  83c001      add eax, 1
0x080496f2  05c0c00408 add eax, 0x804c0c0
; Load into eax a dword from instruction buffer and save into tmp var
0x080496f7  8b00        mov eax, dword [eax]
0x080496f9  8945f0      mov dword [ebp - 0x10], eax

```

```

; Test the var at [ebp-0x28] with 0, this var its the Eflags
0x080496fc  837dd800    cmp dword [ebp - 0x28], 0
; This je is the "op-code"
0x08049700  7408        je 0x804970a
; if value at ebp-28 dont is 0 increment the vIP by 5.
0x08049702  8b45cc      mov eax, dword [ebp - 0x34]
0x08049705  83c005      add eax, 5
0x08049708  eb03        jmp 0x804970d
; if the value was 0 the load the temp var with destination address
0x0804970a  8b45f0      mov eax, dword [ebp - 0x10]
; set te eax value into vIP
0x0804970d  8945cc      mov dword [ebp - 0x34], eax

```

At previous code we can observe how the code extracts next dword from instruction buffer and save into temp variable, then the code does a “cmp dword [ebp - 0x28], 0” and “je 0x804970a”, with these instructions the VM is checking for a value stored into [ebp-0x28] and if this value is 0, it sets *Virtual Instruction Pointer* ([ebp-0x34]) with value saved into [ebp-0x10] (this value is extracted from execution buffer). Otherwise it increments the *Instruction pointer* in 5 and returns the control to main dispatch routine.

This code reveals functionality for [ebp-0x28], it's used by VM to store the result of logic operation. In other words it's the *Virtual Eflags Register*.

Finally take a look into “cmp” instruction to ensure this stuff, if an instruction is involved into *Eflags* be sure is this. “pd 8@0x080499ed”:

```

0x080499ed  90          nop
0x080499ee  8b45e4      mov eax, dword [ebp - 0x1c]
0x080499f1  8b55e0      mov edx, dword [ebp - 0x20]
0x080499f4  89d1        mov ecx, edx
0x080499f6  29c1        sub ecx, eax
0x080499f8  89c8        mov eax, ecx
0x080499fa  8945d8      mov dword [ebp - 0x28], eax
0x080499fd  e965020000 jmp 0x8049c67

```

This code manages distinct addressings, only show the main part. This is the final part of “cmp” and it is doing a “sub”. The *x86 sub* instruction is harder involved at *Eflags* manipulation, basically a cmp and a sub affect at same manner the *x86 Eflags registers*, but in this case the VM use a clever method. It subtracts the two values targeted by instruction parameters in base their addressing, and it stores the value into [ebp-0x28], doing this they only need check later this value and take the correct decision “jne”, “je”, “ja”, “jb” ...

Well I hope you understand all of this ;)

Now it only remains to repeat the analysis process with all the values of the op-code's table and you'll get all the instruction set and special registers involved into VM architecture.

And with all of this at your hand you can make a disassembler for this VM ;)

. Virtual Machine Instruction Set .

Here is the complete instruction set

#	Name	Representation	Len	0	1	2	4
00	NOP	nop	1				
01	RET	ret	1				
02	ADD	addressing1	5/8/8/11	$r0=r1+r2$	$r0=r1+i32_2$	$r0=i32_1+r2$	$r0=i32_1+i32_2$
03	SUB	addressing1	5/8/8/11	$r0=r1-r2$	$r0=r1-i32_2$	$r0=i32_1-r2$	$r0=i32_1-i32_2$
04	MUL	addressing1	5/8/8/11	$r0=r1*r2$	$r0=r1*i32_2$	$r0=i32_1*r2$	$r0=i32_1*i32_2$
05	DIV	addressing5	6/9/9/12				
06	XOR	addressing1	5/8/8/11	$r0=r1^r2$	$r0=r1^i32_2$	$r0=i32_1^r2$	$r0=i32_1^i32_2$
07	NEG	$r0=\text{neg } r1$	3				
08	NOT	$r0=\text{not } r1$	3				
09	AND	addressing1	5/8/8/11	$r0=r1\&r2$	$r0=r1\&i32_2$	$r0=i32_1\&r2$	$r0=i32_1\&i32_2$
10	OR	addressing1	5/8/8/11	$r0=r1 r2$	$r0=r1 i32_2$	$r0=i32_1 r2$	$r0=i32_1 i32_2$
11	SETZ	$r0=r1==0$	3				
12	ROL	addressing1	5/8/8/11	$r0=r1\ll r2$	$r0=r1\ll i32_2$	$r0=i32_1\ll r2$	$r0=i32_1\ll i32_2$
13	ROR	addressing1	5/8/8/11	$r0=r1\gg r2$	$r0=r1\gg i32_2$	$r0=i32_1\gg r2$	$r0=i32_1\gg i32_2$
14	JMP	jmp i32_0	5				
15	CALL	call i32_0	5				
16	JZ	jz i32_0	5				
17	JS	js i32_0	5				
18	JBE	jbe i32_0	5				
19	JG	jg i32_0	5				
20	JNS	jns i32_0	5				
21	JNZ	jnz i32_0	5				
22	AND	addressing2	4/7/7/10	$r1 \& r2$	$r1 \& i32_2$	$i32_1 \& r2$	$i32_1 \& i32_2$
23	CMP	addressing2	4/7/7/10	cmp r1,r2	cmp r1,i32_2	cmp i32_1,r2	cmp i32_1,i32_1
24	MOV	addressing3	4/7	mov r1,r2	mov r1,i32_2		
25	++	$r0++$	2				
26	--	$r0--$	2				
27	MOV	mov r1,[r2]	3	M r,[r]			
28	MOV	Mov [r1],r2	3	M [r],r			
29	ENDVM		1				
30	PUSH	addressing4	3/6	push r1	Push i32_1		
31	POP	pop r0	2				
32	NCALL	call idx	2				

Take a look at “*Representation*” column, after I've finalized the analysis I've found five different types of “addressing” with this lengths:

- *Addressing1*: 5/8/8/11, it is what I've analyzed previously and it uses always a destination register pointed by first byte after the addressing byte.
- *Addressing2*: 4/7/7/10, Same as above but it doesn't use a destination register, instead it stores the result into virtual *Eflag* register. Be aware of there is a special opcode “AND” to set *Eflags* with this addressing mode, it's not the same of binary operation AND.
- *Addressing3*: 4/7, It's only used by a type of “mov”, it manages a *r, r* and *r, im32*
- *Addressing4*: 3/6, It's only used by “push”, it manages *r* and *im32*.
- *Addressing5*: 6/9/9/12. Same as first, but it uses one more byte to select the register where to store the remainder of the div operation (it's only used by div op-code).

Remember how the instruction buffer looks:

[op-code] (addressing)

Where addressing can be present only if the op-code requires for it.

Let me show the first instructions of this VM, as bytes:

“px 256@0x0804d0c0”:

```
[0x0804cfd4]> px 256@0x0804d0c0
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x0804d0c0 1801 003a 1000 0018 0101 db1e 0000 1800 .....
0x0804d0d0 0300 1801 0542 cf74 011b 0403 0600 0404 .....B.t.....
0x0804d0e0 051c 0304 0201 0303 0400 0000 1700 0103 .....
0x0804d0f0 1419 1000 000e 3a10 0000 4c0f 6f01 42ef .....L.o.B.
0x0804d100 7400 62ce 7521 40ce 5402 43ef 7000 62ca t.b.u!@.T.C.p.b.
0x0804d110 7521 44ce 5406 43ef 7c00 62c6 7521 48ce u!D.T.C.|.b.u!H.
0x0804d120 540a 43ef 7800 62c2 7521 4cce 540e 43ef T.C.x.b.u!L.T.C.
0x0804d130 6400 55cf 7401 52b3 6401 42ce 6921 53ce d.U.t.R.d.B.i!S.
0x0804d140 6a01 5cd1 741c 5acf 6a00 46cf 741f 42cd j.\.t.Z.j.F.t.B.
0x0804d150 7501 42c7 7401 42d7 741c 42ef 6517 42cf u.B.t.B.t.B.e.B.
0x0804d160 7411 80df 7401 5acf 701c 4fce 7005 41cf t...t.Z.p.O.p.A.
0x0804d170 7401 5ecf 7003 43cf 7409 42cf 741e 5fd0 t.^.p.C.t.B.t._.
0x0804d180 6a00 5fd7 7501 01cf 7401 4df0 6401 42d7 j._.u...t.M.d.B.
0x0804d190 7501 2dcf 7401 4df0 6401 42d7 7501 2ccf u.-.t.M.d.B.u.,.
0x0804d1a0 7401 4df0 6401 42d7 7501 25cf 7401 4df0 t.M.d.B.u.%.t.M.
0x0804d1b0 6401 42d7 7501 30cf 7401 4df0 6401 42d7 d.B.u.O.t.M.d.B.
```

These values are in hex mode, remember it, to avoid op-code confusions, the first value is 0x18 (24), looking at table reveals this is a:

24	MOV	addressing3	4/7	mov r1,r2	mov r1,i32_2
----	-----	-------------	-----	-----------	--------------

Then looking the next byte, 0x01, it tells us that the size of this mov is 7.

Let me show it using colors:

- offset -	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x0804d0c0	18	01	00	3a	10	00	00	18	01	01	db	1e	00	00	18	00
0x0804d0d0	03	00	18	01	05	42	cf	74	01	1b	04	03	06	00	04	04B.t.....

Red is the op-code. MOV

Yellow is the addressing. r1, i32_1

Green is index to reg, based in addressing byte, r_00.

Blue is the imm32, value 0x0000103a.

This is a “mov r_00,0x0000103a”, followed by another “mov r_01,0x00001edb”, and another with addressing 0: “mov r_03,r_00”.

With this little example you can see how to decode the VM using the table is really easy. But would be much better build a utility to get all the code for us.

. Building an Interpreter .

Now its time to build a little tool which assist to get all the Virtual Code. This is my code for a simple interpreter: [\[decoder.c\]](#)



The code is quite simple, it only has one function to extract and format the parameters from execution buffer and return *op-code* size in base its addressing.

A big switch/case is used to handle each *op-code*, then call to the function previously mentioned to get parameters and size and print the output.

Now is only needed to extract “*the execution buffer*” from the binary, and use the interpreter tool. I ripped the full context of the Virtual Machine from the binary and saved into a file named “vm.code”. To do the job *r2* is the best tool, because it can be used as a hex file editor. Thinking a bit come to mind the address of VM context “0x804C0C0” and then is just needed a dump from this address.

Fire your *r2* and type “s 0x804C0C0” and “v” to go into Visual Mode:

```
[0x0804c0c0 368 baleful]> x @ section..data+128 # 0x804c0c0
- offset -    0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x0804c0c0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c0d0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c0e0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c0f0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c100 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c110 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c120 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c130 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c140 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c150 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c160 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c170 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c180 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c190 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c1a0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c1b0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c1c0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c1d0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c1e0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c1f0 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0804c200 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

Use the “AvPag Key” to inspect theses bytes and get the full Virtual Machine context (first instructions are really at virtual address 0x1000 of the VM):

```
[0x0804dec0 322 baleful]> x @ section..data+7808 # 0x804dec0
- offset -    0 1 2 3 4 5 6 7 8 9 A B C D 0123456789ABCD
0x0804dec0 7401 42c0 4b11 42cf 6c00 42a0 7401 42c0 t.B.K.B.l.B.t.
0x0804dece 42c0 4b11 42cf 6c00 42a1 7401 42c0 B.K.B.l.B.t.B.
0x0804dedc 4b11 42cf 6c00 42a8 7401 42c0 4b11 K.B.l.B.t.B.K.
0x0804deea 42cf 6c00 42ef 7401 42c0 4b11 42cf B.l.B.t.B.K.B.
0x0804def8 6c00 42bf 7401 42c0 4b11 42cf 6c00 l.B.t.B.K.B.l.
0x0804df06 42ae 7401 42c0 4b11 42cf 6c00 42bc B.t.B.K.B.l.B.
0x0804df14 7401 42c0 4b11 42cf 6c00 42bc 7401 t.B.K.B.l.B.t.
0x0804df22 42c0 4b11 42cf 6c00 42b8 7401 42c0 B.K.B.l.B.t.B.
0x0804df30 4b11 42cf 6c00 42a0 7401 42c0 4b11 K.B.l.B.t.B.K.
0x0804df3e 42cf 6c00 42bd 7401 42c0 4b11 42cf B.l.B.t.B.K.B.
0x0804df4c 6c00 42ab 7401 42c0 4b11 42cf 6c00 l.B.t.B.K.B.l.
0x0804df5a 42ee 7401 42c0 4b11 42cf 6c00 42c5 B.t.B.K.B.l.B.
0x0804df68 7401 42c0 4b11 42cf 6c00 42cf 7401 t.B.K.B.l.B.t.
0x0804df76 42d0 7f1e 48d0 7d00 5acf 740a 4d66 B...H.}.Z.t.Mf
0x0804df84 6601 42d7 7501 42cf 7401 5dc4 6b0b f.B.u.B.t.].k.
0x0804df92 5dc6 751c 5fd7 6c19 0000 0000 0000 ].u._.l.....
0x0804dfa0 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfae 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfbc 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfca 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfd8 0000 0000 0000 0000 0000 0000 0000 .....
```

Now we can determine the block size to dump from 0x804c0c0 until 0x804dfd8, a total of 0x1f18 bytes. Using the next *r2* commands we can dump this block to a file.

“s 0x0804c0c0” and “wt vm.code.r2 0x1f18”

```
0x0804df8a 42cf 7401 5dc4 6b0b 5dc6 751c 5fd7 B.t.].k.].u._.
0x0804df98 6c19 0000 0000 0000 0000 0000 0000 l.....
0x0804dfa6 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfb4 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfc2 0000 0000 0000 0000 0000 0000 0000 .....
0x0804dfd0 0000 0000 0000 0000 0000 0000 0000 .....
[0x0804c0c0]> wt vm.code 0x1f18
dumped 0x1f18 bytes
Dumped 7960 bytes from 0x0804c0c0 into vm.code
[0x0804c0c0]>
```

Now with the *vm context file* and the *encode.exe* we can retrieve a representation of the virtual code.

The first virtual code instructions are a decryption routine to decrypt the next bytes at buffer execution, my vm.code sent to my mate thEpOpE is fully decrypted

SkUaTeR

Bonus Stage: A r2 interpreter plug-in.

Radare2 is a great toolbox for RE analysis and it has tons of functionalities and a powerful plug-in system. To implement an “asm” plug-in to interpreter this Virtual Machine is a real joke, and we can recycle the previous C code.


R2 only needs a single file to implement this simple plug-in. But first we need to know how is a basic “asm” plug-in structure. The best way to learn about this is into the *r2* blog at <http://radare.today>.

A month ago an article about how extend *r2* with plug-ins was released “<http://radare.today/extending-r2-with-new-plugins/>”. You can check it to get more details about this. But basically a plug-in must be defined as:

```
static int disassemble(RAsm *a, RAsmOp *op, ut8 *buf, ut64 len) {
    /* TODO: Implement disassemble code here,
     * give a look to the other plugins */
}

RAsmPlugin r_asm_plugin_myarch = {
    .name = "MyArch",
    .desc = "disassembly plugin for MyArch",
    .arch = "myarch",
    .bits = (int[]){ 32, 64, 0 }, /* supported wordsizes */
    .init = NULL,
    .fini = NULL,
    .disassemble = &disassemble,
    .modify = NULL,
    .assemble = NULL,
};

#ifdef CORELIB
struct r_lib_struct_t radare_plugin = {
    .type = R_LIB_TYPE_ASM,
    .data = &r_asm_plugin_myarch
};
#endif
```

And using the previous code with a little modification give us this full plug-in code [[r2 plug-in code](#)] 

It's time to compile it, but we're going to compile it as dynamic plug-in, by this way it's only needed to copy the plug-in into *r2* plug-in directory and start to view the VM code into this great tool. To do the compilation we need have installed a *r2* from *git* to link the code with all their libraries. And compile the code by this way:

```
gcc -shared -fpic asm_baleful.c -o asm_baleful.so $(pkg-config --cflags --libs r_asm)
```

You need have installed *pkg-tools* into your system, and they do all the job related to tell *gcc* the linker options. The compilation process ends too quickly and result a “asm_baleful.so”, now it's only needed to copy this plug-in into *r2* plug-in

directory, you can determine where is located this directory executing `r2` with “-hh” parameter, “`r2 -hh`”:

```
...
Plugins:
plugins  /usr/lib/radare2/last
user     ~/.config/radare2/plugins
LIBR_PLUGINS /usr/lib/radare2/0.9.9-git
```

Then copy the `asm_baleful.so` into “`~/.config/radare2/plugins`” (you need create the directory if it does not exist). And now you have a `r2` with our new plug-in installed. Execute `r2` with the `vm.code`, “`r2 vm.code`”:

```
root@vmkalix86:~/picocftf/dbgsrv# r2 vm.code
-- Ask not what r2 can do for you - ask what you can do for r2
[0x00000000]> s 0x1000
[0x00001000]> e asm.arch=baleful
[0x00001000]> pd 10
0x00001000 1801003a100. mov r_00 0x103a
0x00001007 180101db1e0. mov r_01 0x1edb
0x0000100e 18000300    mov r_03 r_00
0x00001012 18010542cf7. mov r_05 0x174cf42
0x00001019 1b0403     mov r_04 r_03
0x0000101c 0600040405 r_04 = r_04 ^ r_05
0x00001021 1c0304     mov r_03 r_04
0x00001024 02010303040. r_03 = r_03 - 0x0004
0x0000102c 17000103   r_01 cmp r_03
0x00001030 1419100000 jns 0x1019
[0x00001000]>
```

You can see how the amazing `r2` shows the interpretation of virtual code. And can even use the visual mode to inspect and analyze the virtual code.

That's all about VM analysis and implementation a disassembler for it. I hope you'd enjoy about read it and still doing with my mate “thEpOpE”. He continues explaining how was made the analysis of the VM code. It has been a pleasure, c ya soon.

I want to send special greetings to Pancacke (@trufae & @radareorg) by his advice and support with his great tool r2.


SkUaTeR

Punk
Not
Dead.

. Virtual Code - Code Analysis - Virtual Analysis .

. Choose: Fright or 0xD3AD .

While SkUaTeR was making the disassembler, and disassembling all the virtual code, I've been analyzing the `ncall` function. This *op-code* has a table of offsets where it jumps depending on the index given by the parameter. This op-code is mainly used to expand new features to the VM. In this VM there are 17 native functions.

Once SkUaTeR have got all the virtual machine decoded, he sent me it as a plain text file. Yes, no syntax highlighting still. See the [[virtual code](#)] 

And I could take two ways:

Fright : As we had identified the *-instruction pointer*, we could run the virtual code in the VM and stop the virtual program at any desired address. Just setting a conditional breakpoint at the address where the VM loads next op-code. We could set a condition with the value of the instruction pointer, or a condition with the value of the read op-code.

0xD3AD : The other way, is to try understand what the hell this virtual code does. And this was the hard way I've took: to analyze *d3ad c0d3*.

The virtual code starts running a virtual address 0x1000, and in the initial state of the VM, from 0x0000 to 0x0FFF are cleared with 0x00. (All address references I'll do, will be virtual address).

Here is the first instructions of the virtual code:

```
1000:  mov r_00,0x103a
1007:  mov r_01,0x1edb
100e:  mov r_03,r_00
1012:  mov r_05,0x174cf42
1019:  mov r_04,[r_03]
101c:  r_04=r_04 ^ r_05
1024:  mov [r_03],r_04
1028:  r_03=r_03 + 0004
102c:  cmp r_01,r_03
1030:  jns 0x1019
1035:  jmp 0x103a
```

There is a loop, at 1030 there is a conditional jump to 1019, and there is a register that is added with 4. Well, this loop is a snippet that does a xor decode. `r_00` is a pointer with encrypted code, that is copied to `r_03` and incremented later, and `r_01` is the last address with encrypted code. `r_05` gets a 32 bit value that is used to xor the values encrypted that are read at 1019, and are written again, once xored through `r_04`, at 1024.

Cool, a little encryption trick in the beginning. Once the loop ends, it jumps to the start of the code decrypted right now, at offset 103a. The instruction at this address is a `jmp`, (`jmp 0x1bc0`) that jumps to 0x1bc0.

Here is the code at 0x1bc0:

```
1bc0: push r_09
1bc3: push r_0a
1bc6: push r_0b
1bc9: call 0x1a6d
1bce: mov r_01,0x001e
1bd5: mov r_00,0x0004
1bdc: call 0x1080
1be1: mov r_0b,r_00
1be5: jmp 0x1bef
```

It seems really familiar code. Some pushes in the beginning, and some calls. Well, all the code doesn't use many registers, but it's important to notice that `r_00`, `r_01`, and `r_1e` are often used as temporary registers to do other calculations.

Let's see what is at 1a6d:

```
1a6d: mov r_00,0x0050
1a74: call 0x103f
1a79: mov r_00,0x006c
1a80: call 0x103f
1a85: mov r_00,0x0065
1a8c: call 0x103f
1a91: mov r_00,0x0061
1a98: call 0x103f
1a9d: mov r_00,0x0073
1aa4: call 0x103f
1aa9: mov r_00,0x0065
1ab0: call 0x103f
1ab5: mov r_00,0x0020
1abc: call 0x103f
1ac1: mov r_00,0x0065
[...]
1ba5: mov r_00,0x003a
1bac: call 0x103f
1bb1: mov r_00,0x0020
1bb8: call 0x103f
1bbd: ret
```

Here is a piece of code with 2 instructions repeated once and again until the end of this subroutine, that ends with a `ret` op-code. As you may have noticed, all values that are loaded into `r_00`, before the calling to 103f, are ASCII chars. This values composes the string: "Please enter your password: ". Without watching at 103f we can imagine that is the subroutine that prints a char.

```
103f: ncall 0x00 ;putchar
1041: ret
```

In the analysis of this op-code, we have identified and named all the functions in the table of this “native call”. The 0x00 function is `Putchar`. In the virtual code there is a little “import table”, referencing with this technique up to 17 native functions.

```

1042: ncall 0x01
1044: ret
1045: ncall 0x02
1047: ret
1048: ncall 0x03
104a: ret
104b: ncall 0x04
104d: ret
104e: ncall 0x05
1050: ret
[...]
```

This is the table of “native calls” (`ncall`):

<i>ncall #</i>	<i>Name</i>	<i>Operation</i>	<i>Type</i>
0x00	Putchar	putchar(r_00)	Output
0x01	PrintInt	fprintf ("%d", r_00)	Output
0x02	PrintHex	fprintf ("%x", r_00)	Output
0x03	PrintFloat	fprintf ("%f", r_00)	Output
0x04	Getchar	r_00 = getchar()	Input
0x05	ScanInt	r_00 = fscanf ("%d")	Input
0x06	ScanHex	r_00 = fscanf ("%x")	Input
0x07	ScanFloat	r_00 = fscanf ("%f")	Input
0x08	Eof	r_00 = (getchar == EOF ? 1 : 0)	Input
0x09	Int2Float	r_00 (Bin32) → r_00 (IEEE 754)	Fpu
0x0a	Float2Int	r_00 (IEEE 754) → r_00 (Bin32)	Fpu
0x0b	FAdd	r_00 = r_00 + r_01	Fpu
0x0c	FSub	r_00 = r_00 - r_01	Fpu
0x0d	FMul	r_00 = r_00 * r_01	Fpu
0x0e	FDiv	r_00 = r_00 / r_01	Fpu
0x0f	NNop	r_00 = r_00	Misc
0x10	IsNotZero	R_00 = (r_00 != 0 ? 1 : 0)	Misc
0x11	SwapAndAdd	R_00 ↔ RamBaseFree : RamBaseFree += r_00	Misc

Once the string is printed, we can figure out that may be a subroutine to get the password typed by the user. Let's continue with the original flow...

```
1bc6: push r_0b
1bc9: call 0x1a6d ;Prints "Please enter your password: "
1bce: mov r_01,0x001e
1bd5: mov r_00,0x0004
1bdc: call 0x1080
1be1: mov r_0b,r_00
1be5: jmp 0x1bef
```

A new call is done, this time to 1080:

```
1080: push r_1e
1083: push r_1d
1086: mov r_1e,r_01
108a: r_00 = r_1e * r_00
108f: r_00=r_00 +0008
1097: mov r_1d,r_00
109b: ncall 0x11
10a1: jz 0x10c2
10a6: mov r_04,r_1d
10aa: r_04 = r_04 >> 0003
10b5: mov [r_00],r_04
10b5: r_00=r_00 +0008
10bd: pop r_1d
10bf: pop r_1e
10c1: ret
10c2: endvm
```

This snippet is really strange to analyze without running. The `ncall 0x11` is a native function that I've called "SwapAndAdd". This native function uses an initial variable with `0x10000`. This variable is swapped with `r_00`, and then adds `r_00` to the new value of the variable. Every time is called the variable is incremented in the value of `r_00`. For example, if the variable is `0x1010` and `r_00` is `0x30`, `ncall 0x11` will produce that `r_00` is `0x1010` and the variable `0x1040`. If we now does `ncall 0x11` and `r_00` is `0x60`, will produce that `r_00` is `0x1040` and the variable will be `0x10a0`.

The `jz` will never jump because `r_00` and `r_01` aren't 0, but if this `ncall` fails, the code jumps to `10c2`, where there's an `endvm` opcode (the VM ends). This code is something like calculating the necessary space, writing size or pointer to new space available. Let's think about this function as an "AllocateMemory", where the arguments passed are the length, and the size of the elements. The most important thing in this code is to realize that `r_00` is updated before it returns.

When this function ends, the code follows on `1be1`, where the register `r_0b` loads the value of `r_00` (the updated value returned by `1080`). This point is important, as `r_0b` will be relevant later. Finally it jumps to `1bef`.

```
1bef: mov r_09,0x0000
1bf6: jmp 0x1d66
1bfb: mov r_1e,r_09
1bff: r_1e = r_1e * 0004
1c07: mov r_00,r_0b
1c0b: r_1e=r_1e + r_00
1c10: mov r_0a,r_1e
1c14: call 0x104b ; GetChar
1c19: mov r_01,r_00
1c20: mov [r_0a],r_01
1c20: mov r_01,[r_0a]
1c23: mov r_1e,r_01
1c27: mov r_00,0x000a
1c2e: cmp r_1e,r_00
1c32: jz 0x1c3c ; jumps to print sorry..
1c37: jmp 0x1d5e
1c3c: mov r_00,0x0053 ; "Sorry, wrong password!\n"
1c43: call 0x103f
1c48: mov r_00,0x006f
1c4f: call 0x103f
[...]
1d4b: call 0x103f
1d50: mov r_00,0x0000
1d57: pop r_0b
1d59: pop r_0a
1d5b: pop r_09
1d5d: ret
1d5e: r_09=r_09 +0001
1d66: mov r_1e,r_09
1d6a: mov r_00,0x001e
1d71: cmp r_1e,r_00
1d75: js 0x1bfb
```

Previously I've identified the "import table", so I know that function at 104b is a `ncall 0x04`, that is a native call to `getchar`. The code from 1c3c to 1d5d is known too, it's the way to print a string (this time "Sorry, wrong password!\n"); so I've commented this in the code. All the other code is just a *while loop*. The `r_09` register is initialized with 0, and jumps to 1d66, where it is the loop condition. Be careful to don't confuse `r_1e` and the value 0x001e :) The condition uses `r_1e` and `r_00` as temporal registers to do the check, but really it's testing that `r_09` is less than 0x001E. If the condition is true, jumps to the body of the *while loop* (1bfb).

The body is easy, just calls the `getchar` function through the import table, and stores the returned value into the memory pointed by the `r_0a` register. This register is calculated using `r_09` and `r_0b` (again it uses temporal registers `r_1e` and `r_00`).

The position where the value is stored is: `r_0b+(r_09*4)`. The multiplication is needed, because the chars are being stored as 32bits values. With this information, we can assume that `r_0b` is the base pointer for the string that is input by the user, and `r_09` is a counter. Furthermore, the `r_09` register is increased in 1 at 1d5e, right before the *while condition*. The register `r_0b` was calculated by the previous analyzed function at 1080, that we thought as "AllocateMemory".

There is a test in 1c27, that just get value returned by `getchar`, and checks that it's not 0x0a (“\n”). This way the message “Sorry, wrong password!\n” will be printed if the user enters a password with a length less than 1e (30).

Once the condition fails, the *while* loop ends, and the `r_0b` register still has the base pointer of the password typed by the user. After the *while* ends, this is the code:

```
1d7a: call 0x104b ; GetChar
1d7f: mov r_01,r_00
1d83: mov r_1e,r_01
1d87: mov r_00,0x000a
1d8e: cmp r_1e,r_00
1d92: jnz 0x1d9c ; prints sorry
1d97: jmp 0x1ebe
1d9c: mov r_00,0x0053 ; "Sorry, wrong password!\n"
1da3: call 0x103f
[...]
```

```
1ebd: ret
1ebe: mov r_00,r_0b
1ec2: call 0x12a9
1ec7: mov r_00,0x0000
1ece: pop r_0b
1ed0: pop r_0a
1ed2: pop r_09
1ed4: ret
```

Again the `getchar` function is called, this time is to check that the user enters a 0x0a char (“\n”), but this char is not stored anywhere. The code at 1d9c, where jumps if the char is not 0x0a, again is a snippet with a string to print. And yes, again is the string “Sorry, wrong password!\n”. Well, this is the way that code checks the password has a right length of 1e chars. No more, no less.

When all the length checks are passed, the code lands at 1ebe. Here the register `r_00` is loaded with the value of `r_0b` (remember, this register is loaded with the base pointer to the typed password), and calls to 12a9. Hm, it looks like a *CheckPassword* function :)

. How become a hater of temporary registers .

The beginning of the *CheckPassword* function is very familiar at this point of the analysis:

```
12a9: push r_09
12ac: push r_0a
12af: mov r_0a,r_00
12b3: mov r_01,0x001e
12ba: mov r_00,0x0004
```

```
12c1: call 0x1080 ; AllocateMem
12c6: mov r_09,r_00
12ca: mov r_01,0x0004
12d1: mov r_00,0x0004
12d8: call 0x1080 ; AllocateMem
12dd: mov r_05,r_00
```

First, it saves the registers `r_09` and `r_0a`, and loads `r_0a` with the argument passed to this function. That is: from now, `r_0a` is the base pointer to the typed password by the user. Again the “AllocateMem” function is called, this time with different arguments. At `12c1` is called with `1e` and `04`, so we assume that it is reserving `1e` dwords. The address returned in `r_00` is loaded into `r_09`. And the second calling is done using `04` and `04` as arguments. The address returned is loaded into `r_05`. Quickly the code will start to use the value of `r_05`.

```
12e1: mov r_1e,0x00fd
12eb: mov [r_05],r_1e
12eb: mov r_1e,0x0001
12f2: r_1e = r_1e * 0004
12fa: mov r_00,r_05
12fe: r_1e=r_1e + r_00
1303: mov r_01,r_1e
1307: mov r_1e,0x000e
1311: mov [r_01],r_1e
1311: mov r_1e,0x0001
1318: r_1e = r_1e * 0008
1320: mov r_00,r_05
1324: r_1e=r_1e + r_00
1329: mov r_01,r_1e
132d: mov r_1e,0x0063
1337: mov [r_01],r_1e
1337: mov r_1e,0x0001
133e: r_1e = r_1e * 000c
1346: mov r_00,r_05
134a: r_1e=r_1e + r_00
134f: mov r_01,r_1e
1353: mov r_1e,0x004f
135d: mov [r_01],r_1e
```

This piece is using the `r_1e` register as temporary store. So don't be confused with it. This is what is done: first store `00fd` at `[r_05]`. Now loads `r_1e` with `1`, and using `r_00` and `r_01` calculates: $(1*4)+r_05$. It stores this address in `r_01`, and using `r_1e` as temporary register, it saves value `0e` at this address.

Next instructions are very similar to these. It uses `r_1e`, `r_01`, and `r_00` as temporary registers to make same multiplication. But now multiplies by `08`, and `0c`, and always it uses the value of `r_05` as a fix value to add.

Easy to understand. The `r_05` is the base pointer of an array of 4 elements, of 32 bits each one, and the multiplications is calculating the offset where every element

of the array goes. So all this code is initializing the array with this values: [fd, 0e, 63, 4f]. Let's name this array "fourarray".

After this, there is a lot of instructions very similar to these. But this time it uses r_09 as fixed value, instead r_05. It has all sense, because in the start of the function, the AllocateMem was called two times. First time the returned r_00 was loaded into r_09. This time we know that the function was called using 1e and 04 as parameters. This is the code that follows:

```

135d:  mov r_1e,0x008d
1367:  mov [r_09],r_1e
1367:  mov r_1e,0x0001
136e:  r_1e = r_1e * 0004
1376:  mov r_00,r_09
137a:  r_1e=r_1e + r_00
137f:  mov r_01,r_1e
1383:  mov r_1e,0x006f
138d:  mov [r_01],r_1e
138d:  mov r_1e,0x0001
1394:  r_1e = r_1e * 0008
139c:  mov r_00,r_09
13a0:  r_1e=r_1e + r_00
13a5:  mov r_01,r_1e
13a9:  mov r_1e,0x0000
13b3:  mov [r_01],r_1e
13b3:  mov r_1e,0x0001
13ba:  r_1e = r_1e * 000c
13c2:  mov r_00,r_09
13c6:  r_1e=r_1e + r_00
13cb:  mov r_01,r_1e
13cf:  mov r_1e,0x0024
13d9:  mov [r_01],r_1e
13d9:  mov r_1e,0x0001
13e0:  r_1e = r_1e * 0010
13e8:  mov r_00,r_09
13ec:  r_1e=r_1e + r_00
13f1:  mov r_01,r_1e
13f5:  mov r_1e,0x0098
13ff:  mov [r_01],r_1e
[...]
178f:  mov r_1e,0x0001
1796:  r_1e = r_1e * 0074
179e:  mov r_00,r_09
17a2:  r_1e=r_1e + r_00
17a7:  mov r_01,r_1e
17ab:  mov r_1e,0x0077
17b5:  mov [r_01],r_1e

```

Yes, really looong, but predictable (I've omitted some repetitive parts). Again it is the initialization of an array. This time the array is 1E length, so this is the reason to multiply by 4, 8, 0c, 10, 14, and so on, until 74 ((1e-1)*4=74). Once it ends, the array is initialized with the values that, as above, are loaded using r_1e. Let's call this array "longarray". Here is the initial value of the array:

```
[8d, 6f, 00, 24, 98, 7c, 10, 10, 9c, 60, 07, 10, 8b, 63, 10,
10, 9c, 60, 07, 10, 85, 61, 11, 3c, a2, 61, 0b, 10, 90, 77]
```

Now we have three registers with important information: `r_0a` points to the password typed by the user; `r_05` points to an array of 4 elements; and `r_09` points to the new array of 1e elements.

Next instructions resets to 0 three registers, before it starts a loop, that will be very similar to the *while* we've analyzed above:

```
17b5:  mov r_03,0x0000
17bc:  jmp 0x17c6
17c1:  jmp 0x1878
17c6:  mov r_01,0x0000
17cd:  mov r_02,0x0000
17d4:  jmp 0x1860
```

We could assume that this registers, `r_03`, `r_01`, and `r_02` may be used as counters, or indexes. Let's watch 1860 and previous instructions...

```
17d9:  mov r_1e,r_02
17dd:  r_1e = r_1e * 0004
17e5:  mov r_00,r_0a
17e9:  r_1e=r_1e + r_00
17ee:  mov r_03,r_1e
17f2:  mov r_04,[r_03]
17f5:  r_00 = r_02 / 0004
17f5:  r_03 = r_02 mod 0004
17fe:  mov r_1e,r_03
1802:  r_1e = r_1e * 0004
180a:  mov r_00,r_05
180e:  r_1e=r_1e + r_00
1813:  mov r_03,r_1e
1817:  mov r_03,[r_03]
181a:  r_04=r_04 ^ r_03
181f:  mov r_1e,r_02
1823:  r_1e = r_1e * 0004
182b:  mov r_00,r_09
182f:  r_1e=r_1e + r_00
1834:  mov r_03,r_1e
1838:  mov r_03,[r_03]
183b:  mov r_1e,r_04
183f:  mov r_00,r_03
1843:  cmp r_1e,r_00
1847:  jnz 0x1851
184c:  jmp 0x1858
1851:  mov r_01,0x0001
1858:  r_02=r_02 +0001
1860:  mov r_03,r_01
1864:  mov r_1e,r_02
1868:  mov r_00,0x001e
186f:  cmp r_1e,r_00
1873:  js 0x17d9
```

From 1860 to 1873 is the condition of the *while*. The `r_02` is loaded into `r_1e`, and `r_00` is loaded with fix value 1e, then they are compared at 186f.

While `r_02` is less than `1e` the condition will be true, so the code will jump to `17d9`. Here is `r_1e` used again as a temporary register to do some calculations. First `r_02` is multiplied by 4, then is added (using `r_00`) to `r_0a`; and finally the result is stored into `r_03`. That is: `r_03=r_0a+(r_02*4)`.

We know that `r_0a` is the base pointer of the password typed, so we can assume that `r_02` is the index of the password's char we want to read. At `17f2` the char is read from this address to register `r_04`. So now, `r_04=password[r_02]`

The op-code at `17f5` is special, as SkUaTeR has explained before, the division has two destinations: one register to get the quotient, and other one to get the modulus. In this case is only important the modulus, loaded into `r_03`. So `r_03=r_02 mod 4`.

Instructions from `17fe` to `1813` use as temporary registers `r_00` and `r_1e`. The operations done at this range are: `r_03=r_05+((r_02 mod 4)*4)`.

We know that `r_05` is the base pointer to the array we've called "fourarray", the modulus is done with the register it's being used as index to the password, so this is the way to get cyclic values from 0 to 3 (valid indexes to array "fourarray"). `r_02` is the index for the password's char, and it will get values from 0 to `1d`. So the modulus 4 of these values will be: 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, ...

Finally, it is multiplied by 4, because every element in the array "fourarray" has 32 bits. It's easy to understand, as `r_05` is the base address of the array, this calculation gets the base address of the element with the `r_02` index. Furthermore, the next instruction loads into `r_03` the value in this address, and the instruction at `1817` loads the value of the array's element to `r_03`. At this point we have that `r_04=password[r_02]` and `r_03=fourarray[r_02 mod 4]`.

And here comes the most important thing the check does, the instruction at `181a` does a xor: `r_04=r_04^r_03`, that we can substitute with the previous pseudo-code values:

$$r_04 = password[r_02] \wedge fourarray[r_02 \bmod 4]$$

From `181f` to `1838` there is again a calculation to get the base address of an element in an array. This time uses as base address the register `r_09`, that points to address of "longarray". Again it uses the index in `r_02`, and with a multiplication by 4 gets the base address of the `r_02`th element in the array. Finally loads `r_03` with the value of the element of array. So `r_03=longarray[r_02]`

From `183b` to `1847` compares `r_03` and `r_04`, yes, again it's done by using `r_00` and `r_1e` as temporary registers. If these values don't match, then it jumps to

1051 where `r_01` is set to 1. If the values match, jumps to 1851 where the register `r_02` (the index) is incremented in 1. And the loop continues...

Well, it seems clear to understand. `r_01` is a flag, that will be 1 whenever the check fails, and 0 if the check is right. This flag is copied to `r_03`, before the while condition is done. The check at this point is easy to understand (using `i` as the index to every password char):

```
password[i] ^ fourarray[i mod 4] == longarray[i]
```

Finally there is a conditional jump at 187c. This condition is done with `r_03`, that has the flag of the previous check. If `r_03` isn't zero, it prints "Sorry, wrong password!\n", else "Congratulations!\n".

```
1878:  r_03 and r_03
187c:  jnz 0x1952
1881:  mov r_00,0x0043 ; "C"
1888:  call 0x103f
188d:  mov r_00,0x006f ; "o"
1894:  call 0x103f
1899:  mov r_00,0x006e ; "n"
18a0:  call 0x103f
18a5:  mov r_00,0x0067 ; "g"
18ac:  call 0x103f
18b1:  mov r_00,0x0072 ; "r"
18b8:  call 0x103f
18bd:  mov r_00,0x0061 ; "a"
[...]
```

Now the way to get the password is a breeze, we only have to xor `longarray` with `fourarray` repeated. That is: 8d^fd, 6f^0e, 00^63, 24^4f, 98^fd, 7c^0e, 10^63, ... Here is the result, the appreciated and desired flag for this challenge:

```
packers_and_vms_and_xors_oh_my
```

[thEpOpE](#)
crAck & prAy

Greetz to all amn3s1a_team crew

Our respects to w0pr, Insanity, ReSecurity, w3b0n3s, int3pids, The DHARMA Initiative, and our spaniards crew at dcua and fail0verflow

Follow us [@amn3s1a_team](#) | www.amn3s1a.com

```
C:\> forget what we made _
```

• [Data] •

.decoder.c.

```
// Baleful VM Decoder by SkUaTeR
//
// compile: i586-mingw32msvc-gcc -mconsole decoder.c -odecoder.exe
//
// Need a vm.code at same directory

#include <stdio.h>
#include <windows.h>

// Defines for r2 compatibility
#define ut8 char
#define ut32 unsigned int
#define r_strbuf_setf sprintf

int ae_load_file_to_memory(const char *filename, char **result) {
    int size = 0;
    FILE *f = fopen(filename, "rb");
    if (f == NULL) {
        *result = NULL;
        return -1;
    }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, 0, SEEK_SET);
    *result = (char *)malloc(size+1);
    if (size != fread(*result, sizeof(char), size, f)) {
        free(*result);
        return -2;
    }
    fclose(f);
    (*result)[size] = 0;
    return size;
}

int anal_baleful_getregs(const ut8 *buf, ut8 * b, char * oper, int type) {
    const ut8 * c;
    const ut8 *r0;
    const ut8 *r1;
    const ut8 *r2;
    const ut8 *r3;
    const ut32 *imm;
    const ut32 *imml;

    int size=0;
    c = buf+1;
    switch(type) {
    case 0: // 8 8 11 5
        r0 = buf + 2;
        switch(*c) {
        case 1:
            r1 = buf + 3;
            imm = buf + 4;
            r_strbuf_setf(b, "r_%02x = r_%02x %s 0x%04x", *r0, *r1, oper, *imm);
            size=8;
            break;
        case 2:
            imm = buf + 3;
            r1 = buf + 4;
            r_strbuf_setf(b, "r_%02x = 0x%04x %s r_%02x", *r0, *imm, oper, *r1);
            size=8;
            break;
        case 4:
            imm = buf + 3;

```

```
        imm1 = buf + 7;
        r_strbuf_setf(b, "r_%02x = 0x%04x %s 0x%04x", *r0, *imm, oper, *imm1);
        size=11;
        break;
    case 0:
        r1 = buf + 3;
        r2 = buf + 4;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=5;
        break;
    default:
        r1 = buf + 3;
        r2 = buf + 4;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=5;
        break;
    }
    break;

case 1: // 9 9 12 6
    r0 = buf + 2;
    r3 = buf + 3; // address to save reminder
    switch(*c) {
    case 1:
        r1 = buf + 4;
        imm = buf + 5;
        r_strbuf_setf(b, "r_%02x = r_%02x %s 0x%04x", *r0, *r1, oper, *imm);
        size=9;
        break;
    case 2:
        r1 = buf + 5;
        r_strbuf_setf(b, "r_%02x = 0x%04x %s r_%02x", *r0, *imm, oper, *r1);
        size=9;
        break;
    case 4:
        imm = buf + 4;
        imm1 = buf + 8;
        r_strbuf_setf(b, "r_%02x = 0x%04x %s 0x%04x", *r0, *imm, oper, *imm1);
        size=12;
        break;
    case 0:
        r1 = buf + 4;
        r2 = buf + 5;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=6;
        break;
    default:
        r1 = buf + 4;
        r2 = buf + 5;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=6;
        break;
    }
    break;

case 2: // 7 7 10 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        r_strbuf_setf(b, "r_%02x %s 0x%04x", *r1, oper, *imm);
        size=7;
        break;
    case 2:
        imm = buf + 2;
        r1 = buf + 6;
        r_strbuf_setf(b, "0x%04x %s r_%02x", *imm, oper, *r1);
        size=7;
        break;
    case 4:
        imm = buf + 2;
        imm1 = buf + 6;
```



```
        r_strbuf_setf(b, "0x%04x %s 0x%04x",*imm,oper,*imm1);
        size=10;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "r_%02x %s r_%02x",*r1,oper,*r2);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "r_%02x %s r_%02x",*r1,oper,*r2);
        size=4;
        break;
    }
    break;

case 3:// 7 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        r_strbuf_setf(b, "%s r_%02x,0x%04x",oper,*r1,*imm);
        size=7;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "%s r_%02x,r_%02x",oper,*r1,*r2);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "%s r_%02x,r_%02x",oper,*r1,*r2);
        size=4;
        break;
    }
    break;

case 4: // 6 3
    switch(*c) {
    case 1:
        imm = buf + 2;
        r_strbuf_setf(b, "%s 0x%04x",oper,*imm);
        size=6;
        break;
    case 0:
        r0 = buf + 2;
        r_strbuf_setf(b, "%s r_%02x",oper,*r0);
        size=3;
        break;
    default:
        r0 = buf + 2;
        r_strbuf_setf(b, "%s r_%02x",oper,*r0);
        size=3;
        break;
    }
    break;

case 5: //5
    imm = buf + 1;
    r_strbuf_setf(b, "%s 0x%04x",oper,*imm);
    size=5;
    break;
case 6://2
    r0 = buf + 1;
    r_strbuf_setf(b, "%s r_%02x",oper,*r0);
    size=2;
    break;
break;
```

```
    }
    return size;
}

int main(void)
{
    int size;
    int reip;
    int vIP;
    int tmp;
    char *vmMemoryBase=0;
    int buf;
    char salida[1024];
    const ut8 *r = 0;
    const ut8 *r0 = 0;
    const ut8 *r1 = 0;
    const ut8 *p = 0;
    const ut32 *imm = 0;
    const ut32 *imm1 = 0;
    // Set virtual ip to 0x1000, we have a full context and instruction
    // start at this offset into the context.
    vIP = 0x1000u;
    tmp=vIP;
    size = ae_load_file_to_memory("vm.code", &vmMemoryBase);
    while(vIP<size) {
        buf=&vmMemoryBase[vIP];
        tmp=vIP;
        switch (vmMemoryBase[vIP]) {
            case 2: // 8 8 11 5
                vIP+=anal_baleful_getregs(buf,&salida,"+",0);
                break;
            case 3: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"-",0);
                break;
            case 4: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"*",0);
                break;
            case 6: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"^",0);
                break;
            case 9: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"%",0);
                break;
            case 10: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"|",0);
                break;
            case 12: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"<<",0);
                break;
            case 13: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,">>",0);
                break;
            case 5: // 9 9 12 6
                vIP+= anal_baleful_getregs(buf,&salida,"/",1);
                break;
            case 22: // 7 7 10 4
                vIP+= anal_baleful_getregs(buf,&salida,"and",2);
                break;
            case 23: // 7 7 10 4
                vIP+= anal_baleful_getregs(buf,&salida,"cmp",2);
                break;
            case 24: //7 4

                vIP+= anal_baleful_getregs(buf,&salida,"mov",3);
                break;
            case 30: //6 3
                p = buf + 1;
                vIP+= anal_baleful_getregs(buf,&salida,"push",4);
                break;
            case 15: //5
                imm = buf + 1;
                vIP+= anal_baleful_getregs(buf,&salida,"call",5);
```

```
        break;
case 14: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jmp",5);
    break;
case 16: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jz",5);
    break;
case 17: //5
    vIP+= anal_baleful_getregs(buf,&salida,"js",5);
    break;
case 18: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jbe",5);
    break;
case 19: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jg",5);
    break;
case 20: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jns",5);
    break;
case 21: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jnz",5);
    break;

case 27:
    r = buf + 1;
    r1 = buf + 2;
    vIP+= 3;
    r_strbuf_setf (&salida, "mov r_%02x,[r_%02x]",*r,*r1);
    break;
case 28://0x1c
    r = buf + 1;
    r1 = buf + 2;
    vIP+= 3;
    r_strbuf_setf (&salida, "mov [r_%02x],r_%02x",*r,*r1);
    break;
case 11:
    r_strbuf_setf (&salida, "regX = regY==0");
    vIP+= 3;
    break;
case 7:
    r_strbuf_setf (&salida, "regX = NEG regY");
    vIP+= 3;
    break;
case 8:
    r_strbuf_setf (&salida, "regX = NOT regY");
    vIP+= 3;
    break;
case 25:
    vIP+= anal_baleful_getregs(buf,&salida,"++",6);
    break;
case 26:
    r = buf + 1;
    vIP+= anal_baleful_getregs(buf,&salida,"--",6);
    break;
case 31:
    vIP+= anal_baleful_getregs(buf,&salida,"pop",6);
    break;
case 32:
    p = buf + 1;
    vIP+= 2;
    if (*p==0)
        r_strbuf_setf (&salida, "apicall: putchar()");
    else
        r_strbuf_setf (&salida, "apicall: %02x",*p);
    break;
case 1:
    vIP+= 1;
    r_strbuf_setf (&salida, "ret");
    break;
case 0:
    vIP+= 1;
    r_strbuf_setf (&salida, "nop");
    break;
```

```

        case 29:
            vIP+= 1;
            r_strbuf_setf (&salida, "end virtual");
            break;

        default:
            vIP+= 1;
            r_strbuf_setf (&salida, "nop");
            break;
    }
    printf("%08x: %s    (size = %i)\n",tmp,salida,vIP-tmp);
    getchar();
};

return 0;
}

```

. radare2 plug-in code .

```

/* radare - LGPL - Copyright 2009-2014 - pancake, nibble */
/* baleful plugin by SkUaTeR */

#include <stdio.h>
#include <string.h>
#include <r_types.h>
#include <r_lib.h>
#include <r_asm.h>
int asm_baleful_getregs(const ut8 *buf,ut8 * b,char * oper,int type) {
    const ut8 * c;
    const ut8 *r0;
    const ut8 *r1;
    const ut8 *r2;
    const ut8 *r3;
    const ut32 *imm;
    const ut32 *imm1;

    int size=0;
    c = buf+1;
    switch(type) {
    case 0: // 8 8 11 5
        r0 = buf + 2;
        switch(*c) {
        case 1:
            r1 = buf + 3;
            imm = buf + 4;
            snprintf(b, 64, "r_%02x = r_%02x %s 0x%04x",*r0,*r1,oper,*imm);
            //snprintf(b, 64, "%s",oper);
            size=8;
            break;

        case 2:
            imm = buf + 3;
            r1 = buf + 4;
            snprintf(b, 64, "r_%02x = 0x%04x %s r_%02x",*r0,*imm,oper,*r1);

            //snprintf(b, 64, "%s",oper);
            size=8;
            break;

        case 4:
            imm = buf + 3;
            imm1 = buf + 7;
            snprintf(b, 64, "r_%02x = 0x%04x %s 0x%04x",*r0,*imm,oper,*imm1);
            //snprintf(b, 64, "%s",oper);
            size=11;
            break;

        case 0:

```

```
        r1 = buf + 3;
        r2 = buf + 4;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=5;
        break;
default:
        r1 = buf + 3;
        r2 = buf + 4;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);

        //snprintf(b, 64, "%s",oper);
        size=5;
        break;
    }
    break;
case 1: // 9 9 12 6
    r0 = buf + 2;
    r3 = buf + 3; // guarda aki el resto
    switch(*c) {
    case 1:
        r1 = buf + 4;
        imm = buf + 5;
        snprintf(b, 64, "r_%02x = r_%02x %s 0x%04x",*r0,*r1,oper,*imm);
        //snprintf(b, 64, "%s",oper);
        size=9;
        break;
    case 2:
        r1 = buf + 5;
        snprintf(b, 64, "r_%02x = 0x%04x %s r_%02x",*r0,*imm,oper,*r1);

        //snprintf(b, 64, "%s",oper);
        size=9;
        break;
    case 4:
        imm = buf + 4;
        imm1 = buf + 8;
        snprintf(b, 64, "r_%02x = 0x%04x %s 0x%04x",*r0,*imm,oper,*imm1);
        //snprintf(b, 64, "%s",oper);
        size=12;
        break;
    case 0:
        r1 = buf + 4;
        r2 = buf + 5;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=6;
        break;
    default:
        r1 = buf + 4;
        r2 = buf + 5;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);

        //snprintf(b, 64, "%s",oper);
        size=6;
        break;
    }
    break;
case 2: // 7 7 10 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        snprintf(b, 64, "r_%02x %s 0x%04x",*r1,oper,*imm);
        //snprintf(b, 64, "%s",oper);
        size=7;
        break;
    case 2:
        imm = buf + 2;
        r1 = buf + 6;
        snprintf(b, 64, "0x%04x %s r_%02x",*imm,oper,*r1);
        //snprintf(b, 64, "%s",oper);
```

```
        size=7;
        break;
    case 4:
        imm = buf + 2;
        imm1 = buf + 6;
        snprintf(b, 64, "0x%04x %s 0x%04x",*imm,oper,*imm1);
        //snprintf(b, 64, "%s",oper);
        size=10;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "r_%02x %s r_%02x",*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "r_%02x %s r_%02x",*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    }
    break;
case 3:// 7 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        snprintf(b, 64, "%s r_%02x,0x%04x",oper,*r1,*imm);
        //snprintf(b, 64, "%s",oper);
        size=7;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "%s r_%02x,r_%02x",oper,*r1,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "%s r_%02x,r_%02x",oper,*r1,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    }
    break;
case 4: // 6 3
    switch(*c) {
    case 1:
        imm = buf + 2;
        snprintf(b, 64, "%s 0x%04x",oper,*imm);

        //snprintf(b, 64, "%s",oper);
        size=6;
        break;
    case 0:
        r0 = buf + 2;
        snprintf(b, 64, "%s r_%02x",oper,*r0);

        //snprintf(b, 64, "%s",oper);
        size=3;
        break;
    default:
        r0 = buf + 2;
        snprintf(b, 64, "%s r_%02x",oper,*r0);

        //snprintf(b, 64, "%s",oper);
        size=3;
```

```
        break;
    }
    break;
case 5: //5
    imm = buf + 1;
    snprintf(b, 64, "%s 0x%04x", oper, *imm);

    //snprintf(b, 64, "%s", oper);
    size=5;
    break;
case 6://2
    r0 = buf + 1;
    snprintf(b, 64, "%s r_%02x", oper, *r0);

    //snprintf(b, 64, "%s", oper);
    size=2;
    break;
break;
}
return size;
}

static int disassemble(RAsm *a, RAsmOp *op, const ut8 *buf, int len) {
    const ut8 *p;
    const ut8 *r;
    const ut8 *r1;
    const ut32 *imm;
    const ut32 *imm1;

    switch (*buf) {
        case 2://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "+", 0);
            break;
        case 3://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "-", 0);
            break;
        case 4://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "*", 0);
            break;
        case 6://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "^", 0);
            break;
        case 9://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "&", 0);
            break;
        case 10://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "|", 0);
            break;
        case 12://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "<<", 0);
            break;
        case 13://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, ">>", 0);
            break;
        case 5: // //9 9 12 6
            op->size = asm_baleful_getregs(buf, op->buf_asm, "/", 1);
            break;
        case 22: // 7 7 10 4
            op->size = asm_baleful_getregs(buf, op->buf_asm, "and", 2);
            break;
        case 23: // 7 7 10 4
            op->size = asm_baleful_getregs(buf, op->buf_asm, "cmp", 2);
            break;
        case 24: //7 4
            op->size = asm_baleful_getregs(buf, op->buf_asm, "mov", 3);
            break;
        case 30: // 6 3
            op->size = asm_baleful_getregs(buf, op->buf_asm, "push", 4);
            break;
        case 15: //5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "call", 5);
            break;
    }
```

```
case 14: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jmp",5);
    break;
case 16: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jz",5);
    break;
case 17: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"js",5);
    break;
case 18: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jbe",5);
    break;
case 19: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jg",5);
    break;
case 20: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jns",5);
    break;
case 21: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jnz",5);
    break;
case 27: //3
    op->size = 3;
    r = buf + 1;
    r1 = buf + 2;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "mov r_%02x,[r_%02x]",*r,*r1);
    break;
case 28: //3
    r = buf + 1;
    r1 = buf + 2;
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "mov [r_%02x],r_%02x",*r,*r1);
    break;
case 11: //3
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "regX= regY==0");
    break;
case 7: //3
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "regX= NEG regY");
    break;
case 8: //3
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "regX= NOT regY");
    break;
case 25: //2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"++",6);
    break;
case 26: //2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"--",6);
    break;
case 31: //2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"pop",6);
    break;
case 32: // 2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"apicall",6);
    break;
case 1:
    op->size = 1;
    strcpy (op->buf_asm, "ret");

    break;
case 0:
    op->size = 1;
    strcpy (op->buf_asm, "nop");

    break;
case 29:
    op->size = 1;
    strcpy (op->buf_asm, "end virtual");
    break;
default:
```



```
        op->size = 1;
        strcpy (op->buf_asm, "nop");
        break;
    }
    return op->size;
}

RAsmPlugin r_asm_plugin_baleful = {
    .name = "baleful",
    .arch = "baleful",
    .license = "LGPL3",
    .bits = 32,
    .desc = "Baleful",
    .init = NULL,
    .fini = NULL,
    .disassemble = &disassemble,
    //.assemble = null// &assemble
};

#ifdef CORELIB
struct r_lib_struct_t radare_plugin = {
    .type = R_LIB_TYPE_ASM,
    .data = &r_asm_plugin_baleful
};
#endif
```

. Original Virtual Code .

```
1000: mov r_00,0x103a
1007: mov r_01,0x1edb
100e: mov r_03,r_00
1012: mov r_05,0x174cf42
1019: mov r_04,[r_03]
101c: r_04=r_04 ^ r_05
1024: mov [r_03],r_04
1024: r_03=r_03 +0004
102c: cmp r_01,r_03
1030: jns 0x1019
1035: jmp 0x103a
103a: jmp 0x1bc0
103f: ncall 0x00
1041: ret
1042: ncall 0x01
1044: ret
1045: ncall 0x02
1047: ret
1048: ncall 0x03
104a: ret
104b: ncall 0x04
104d: ret
104e: ncall 0x05
1050: ret
1051: ncall 0x06
1053: ret
1054: ncall 0x07
1056: ret
1057: ncall 0x08
1059: ret
105a: ncall 0x09
105c: ret
105d: ncall 0x0a
105f: ret
1060: ncall 0x0b
1062: ret
1063: ncall 0x0c
```

```
1065: ret
1066: ncall 0x0d
1068: ret
1069: ncall 0x0e
106b: ret
106c: ncall 0x0f
106e: ret
106f: ncall 0x10
1071: ret
1072: cmp r_00,r_00
1076: jz 0x107c
107b: ret
107d: ncall 0x11
107f: ret
1080: push r_1e
1083: push r_1d
1086: mov r_1e,r_01
108a: r_00 = r_1e * r_00
108f: r_00=r_00 +0008
1097: mov r_1d,r_00
109b: ncall 0x11
10a1: jz 0x10c2
10a6: mov r_04,r_1d
10aa: r_04 = r_04 >> 0003
10b5: mov [r_00],r_04
10b5: r_00=r_00 +0008
10bd: pop r_1d
10bf: pop r_1e
10c1: ret
10c2: endvm
10c3: mov r_00,0x0043
10ca: call 0x103f
10cf: mov r_00,0x006f
10d6: call 0x103f
10db: mov r_00,0x006e
10e2: call 0x103f
10e7: mov r_00,0x0067
10ee: call 0x103f
10f3: mov r_00,0x0072
10fa: call 0x103f
10ff: mov r_00,0x0061
1106: call 0x103f
110b: mov r_00,0x0074
1112: call 0x103f
1117: mov r_00,0x0075
111e: call 0x103f
1123: mov r_00,0x006c
112a: call 0x103f
112f: mov r_00,0x0061
1136: call 0x103f
113b: mov r_00,0x0074
1142: call 0x103f
1147: mov r_00,0x0069
114e: call 0x103f
1153: mov r_00,0x006f
115a: call 0x103f
115f: mov r_00,0x006e
1166: call 0x103f
116b: mov r_00,0x0073
1172: call 0x103f
1177: mov r_00,0x0021
117e: call 0x103f
1183: mov r_00,0x000a
118a: call 0x103f
118f: ret
1192: mov r_00,0x0053
1199: call 0x103f
119e: mov r_00,0x006f
11a5: call 0x103f
11aa: mov r_00,0x0072
11b1: call 0x103f
11b6: mov r_00,0x0072
```

```
11bd: call 0x103f
11c2: mov r_00,0x0079
11c9: call 0x103f
11ce: mov r_00,0x002c
11d5: call 0x103f
11da: mov r_00,0x0020
11e1: call 0x103f
11e6: mov r_00,0x0077
11ed: call 0x103f
11f2: mov r_00,0x0072
11f9: call 0x103f
11fe: mov r_00,0x006f
1205: call 0x103f
120a: mov r_00,0x006e
1211: call 0x103f
1216: mov r_00,0x0067
121d: call 0x103f
1222: mov r_00,0x0020
1229: call 0x103f
122e: mov r_00,0x0070
1235: call 0x103f
123a: mov r_00,0x0061
1241: call 0x103f
1246: mov r_00,0x0073
124d: call 0x103f
1252: mov r_00,0x0073
1259: call 0x103f
125e: mov r_00,0x0077
1265: call 0x103f
126a: mov r_00,0x006f
1271: call 0x103f
1276: mov r_00,0x0072
127d: call 0x103f
1282: mov r_00,0x0064
1289: call 0x103f
128e: mov r_00,0x0021
1295: call 0x103f
129a: mov r_00,0x000a
12a1: call 0x103f
12a6: ret
12a9: push r_09
12ac: push r_0a
12af: mov r_0a,r_00
12b3: mov r_01,0x001e
12ba: mov r_00,0x0004
12c1: call 0x1080
12c6: mov r_09,r_00
12ca: mov r_01,0x0004
12d1: mov r_00,0x0004
12d8: call 0x1080
12dd: mov r_05,r_00
12e1: mov r_1e,0x00fd
12eb: mov [r_05],r_1e
12eb: mov r_1e,0x0001
12f2: r_1e = r_1e * 0004
12fa: mov r_00,r_05
12fe: r_1e=r_1e + r_00
1303: mov r_01,r_1e
1307: mov r_1e,0x000e
1311: mov [r_01],r_1e
1311: mov r_1e,0x0001
1318: r_1e = r_1e * 0008
1320: mov r_00,r_05
1324: r_1e=r_1e + r_00
1329: mov r_01,r_1e
132d: mov r_1e,0x0063
1337: mov [r_01],r_1e
1337: mov r_1e,0x0001
133e: r_1e = r_1e * 000c
1346: mov r_00,r_05
134a: r_1e=r_1e + r_00
134f: mov r_01,r_1e
```

```
1353: mov r_1e,0x004f
135d: mov [r_01],r_1e
135d: mov r_1e,0x008d
1367: mov [r_09],r_1e
1367: mov r_1e,0x0001
136e: r_1e = r_1e * 0004
1376: mov r_00,r_09
137a: r_1e=r_1e + r_00
137f: mov r_01,r_1e
1383: mov r_1e,0x006f
138d: mov [r_01],r_1e
138d: mov r_1e,0x0001
1394: r_1e = r_1e * 0008
139c: mov r_00,r_09
13a0: r_1e=r_1e + r_00
13a5: mov r_01,r_1e
13a9: mov r_1e,0x0000
13b3: mov [r_01],r_1e
13b3: mov r_1e,0x0001
13ba: r_1e = r_1e * 000c
13c2: mov r_00,r_09
13c6: r_1e=r_1e + r_00
13cb: mov r_01,r_1e
13cf: mov r_1e,0x0024
13d9: mov [r_01],r_1e
13d9: mov r_1e,0x0001
13e0: r_1e = r_1e * 0010
13e8: mov r_00,r_09
13ec: r_1e=r_1e + r_00
13f1: mov r_01,r_1e
13f5: mov r_1e,0x0098
13ff: mov [r_01],r_1e
13ff: mov r_1e,0x0001
1406: r_1e = r_1e * 0014
140e: mov r_00,r_09
1412: r_1e=r_1e + r_00
1417: mov r_01,r_1e
141b: mov r_1e,0x007c
1425: mov [r_01],r_1e
1425: mov r_1e,0x0001
142c: r_1e = r_1e * 0018
1434: mov r_00,r_09
1438: r_1e=r_1e + r_00
143d: mov r_01,r_1e
1441: mov r_1e,0x0010
144b: mov [r_01],r_1e
144b: mov r_1e,0x0001
1452: r_1e = r_1e * 001c
145a: mov r_00,r_09
145e: r_1e=r_1e + r_00
1463: mov r_01,r_1e
1467: mov r_1e,0x0010
1471: mov [r_01],r_1e
1471: mov r_1e,0x0001
1478: r_1e = r_1e * 0020
1480: mov r_00,r_09
1484: r_1e=r_1e + r_00
1489: mov r_01,r_1e
148d: mov r_1e,0x009c
1497: mov [r_01],r_1e
1497: mov r_1e,0x0001
149e: r_1e = r_1e * 0024
14a6: mov r_00,r_09
14aa: r_1e=r_1e + r_00
14af: mov r_01,r_1e
14b3: mov r_1e,0x0060
14bd: mov [r_01],r_1e
14bd: mov r_1e,0x0001
14c4: r_1e = r_1e * 0028
14cc: mov r_00,r_09
14d0: r_1e=r_1e + r_00
14d5: mov r_01,r_1e
```

```
14d9: mov r_1e,0x0007
14e3: mov [r_01],r_1e
14e3: mov r_1e,0x0001
14ea: r_1e = r_1e * 002c
14f2: mov r_00,r_09
14f6: r_1e=r_1e + r_00
14fb: mov r_01,r_1e
14ff: mov r_1e,0x0010
1509: mov [r_01],r_1e
1509: mov r_1e,0x0001
1510: r_1e = r_1e * 0030
1518: mov r_00,r_09
151c: r_1e=r_1e + r_00
1521: mov r_01,r_1e
1525: mov r_1e,0x008b
152f: mov [r_01],r_1e
152f: mov r_1e,0x0001
1536: r_1e = r_1e * 0034
153e: mov r_00,r_09
1542: r_1e=r_1e + r_00
1547: mov r_01,r_1e
154b: mov r_1e,0x0063
1555: mov [r_01],r_1e
1555: mov r_1e,0x0001
155c: r_1e = r_1e * 0038
1564: mov r_00,r_09
1568: r_1e=r_1e + r_00
156d: mov r_01,r_1e
1571: mov r_1e,0x0010
157b: mov [r_01],r_1e
157b: mov r_1e,0x0001
1582: r_1e = r_1e * 003c
158a: mov r_00,r_09
158e: r_1e=r_1e + r_00
1593: mov r_01,r_1e
1597: mov r_1e,0x0010
15a1: mov [r_01],r_1e
15a1: mov r_1e,0x0001
15a8: r_1e = r_1e * 0040
15b0: mov r_00,r_09
15b4: r_1e=r_1e + r_00
15b9: mov r_01,r_1e
15bd: mov r_1e,0x009c
15c7: mov [r_01],r_1e
15c7: mov r_1e,0x0001
15ce: r_1e = r_1e * 0044
15d6: mov r_00,r_09
15da: r_1e=r_1e + r_00
15df: mov r_01,r_1e
15e3: mov r_1e,0x0060
15ed: mov [r_01],r_1e
15ed: mov r_1e,0x0001
15f4: r_1e = r_1e * 0048
15fc: mov r_00,r_09
1600: r_1e=r_1e + r_00
1605: mov r_01,r_1e
1609: mov r_1e,0x0007
1613: mov [r_01],r_1e
1613: mov r_1e,0x0001
161a: r_1e = r_1e * 004c
1622: mov r_00,r_09
1626: r_1e=r_1e + r_00
162b: mov r_01,r_1e
162f: mov r_1e,0x0010
1639: mov [r_01],r_1e
1639: mov r_1e,0x0001
1640: r_1e = r_1e * 0050
1648: mov r_00,r_09
164c: r_1e=r_1e + r_00
1651: mov r_01,r_1e
1655: mov r_1e,0x0085
165f: mov [r_01],r_1e
```

```
165f: mov r_1e,0x0001
1666: r_1e = r_1e * 0054
166e: mov r_00,r_09
1672: r_1e=r_1e + r_00
1677: mov r_01,r_1e
167b: mov r_1e,0x0061
1685: mov [r_01],r_1e
1685: mov r_1e,0x0001
168c: r_1e = r_1e * 0058
1694: mov r_00,r_09
1698: r_1e=r_1e + r_00
169d: mov r_01,r_1e
16a1: mov r_1e,0x0011
16ab: mov [r_01],r_1e
16ab: mov r_1e,0x0001
16b2: r_1e = r_1e * 005c
16ba: mov r_00,r_09
16be: r_1e=r_1e + r_00
16c3: mov r_01,r_1e
16c7: mov r_1e,0x003c
16d1: mov [r_01],r_1e
16d1: mov r_1e,0x0001
16d8: r_1e = r_1e * 0060
16e0: mov r_00,r_09
16e4: r_1e=r_1e + r_00
16e9: mov r_01,r_1e
16ed: mov r_1e,0x00a2
16f7: mov [r_01],r_1e
16f7: mov r_1e,0x0001
16fe: r_1e = r_1e * 0064
1706: mov r_00,r_09
170a: r_1e=r_1e + r_00
170f: mov r_01,r_1e
1713: mov r_1e,0x0061
171d: mov [r_01],r_1e
171d: mov r_1e,0x0001
1724: r_1e = r_1e * 0068
172c: mov r_00,r_09
1730: r_1e=r_1e + r_00
1735: mov r_01,r_1e
1739: mov r_1e,0x000b
1743: mov [r_01],r_1e
1743: mov r_1e,0x0001
174a: r_1e = r_1e * 006c
1752: mov r_00,r_09
1756: r_1e=r_1e + r_00
175b: mov r_01,r_1e
175f: mov r_1e,0x0010
1769: mov [r_01],r_1e
1769: mov r_1e,0x0001
1770: r_1e = r_1e * 0070
1778: mov r_00,r_09
177c: r_1e=r_1e + r_00
1781: mov r_01,r_1e
1785: mov r_1e,0x0090
178f: mov [r_01],r_1e
178f: mov r_1e,0x0001
1796: r_1e = r_1e * 0074
179e: mov r_00,r_09
17a2: r_1e=r_1e + r_00
17a7: mov r_01,r_1e
17ab: mov r_1e,0x0077
17b5: mov [r_01],r_1e
17b5: mov r_03,0x0000
17bc: jmp 0x17c6
17c1: jmp 0x1878
17c6: mov r_01,0x0000
17cd: mov r_02,0x0000
17d4: jmp 0x1860
17d9: mov r_1e,r_02
17dd: r_1e = r_1e * 0004
17e5: mov r_00,r_0a
```

```
17e9: r_1e=r_1e + r_00
17ee: mov r_03,r_1e
17f2: mov r_04,[r_03]
17f5: r_00 = r_02 / 0004
17f5: r_03 = r_02 mod 0004
17fe: mov r_1e,r_03
1802: r_1e = r_1e * 0004
180a: mov r_00,r_05
180e: r_1e=r_1e + r_00
1813: mov r_03,r_1e
1817: mov r_03,[r_03]
181a: r_04=r_04 ^ r_03
181f: mov r_1e,r_02
1823: r_1e = r_1e * 0004
182b: mov r_00,r_09
182f: r_1e=r_1e + r_00
1834: mov r_03,r_1e
1838: mov r_03,[r_03]
183b: mov r_1e,r_04
183f: mov r_00,r_03
1843: cmp r_1e,r_00
1847: jnz 0x1851
184c: jmp 0x1858
1851: mov r_01,0x0001
1858: r_02=r_02 +0001
1860: mov r_03,r_01
1864: mov r_1e,r_02
1868: mov r_00,0x001e
186f: cmp r_1e,r_00
1873: js 0x17d9
1078: r_03 and r_03
187c: jnz 0x1952
1881: mov r_00,0x0043
1888: call 0x103f
188d: mov r_00,0x006f
1894: call 0x103f
1899: mov r_00,0x006e
18a0: call 0x103f
18a5: mov r_00,0x0067
18ac: call 0x103f
18b1: mov r_00,0x0072
18b8: call 0x103f
18bd: mov r_00,0x0061
18c4: call 0x103f
18c9: mov r_00,0x0074
18d0: call 0x103f
18d5: mov r_00,0x0075
18dc: call 0x103f
18e1: mov r_00,0x006c
18e8: call 0x103f
18ed: mov r_00,0x0061
18f4: call 0x103f
18f9: mov r_00,0x0074
1900: call 0x103f
1905: mov r_00,0x0069
190c: call 0x103f
1911: mov r_00,0x006f
1918: call 0x103f
191d: mov r_00,0x006e
1924: call 0x103f
1929: mov r_00,0x0073
1930: call 0x103f
1935: mov r_00,0x0021
193c: call 0x103f
1941: mov r_00,0x000a
1948: call 0x103f
194d: jmp 0x1a66
1952: mov r_00,0x0053
1959: call 0x103f
195e: mov r_00,0x006f
1965: call 0x103f
196a: mov r_00,0x0072
```

```
1971: call 0x103f
1976: mov r_00,0x0072
197d: call 0x103f
1982: mov r_00,0x0079
1989: call 0x103f
198e: mov r_00,0x002c
1995: call 0x103f
199a: mov r_00,0x0020
19a1: call 0x103f
19a6: mov r_00,0x0077
19ad: call 0x103f
19b2: mov r_00,0x0072
19b9: call 0x103f
19be: mov r_00,0x006f
19c5: call 0x103f
19ca: mov r_00,0x006e
19d1: call 0x103f
19d6: mov r_00,0x0067
19dd: call 0x103f
19e2: mov r_00,0x0020
19e9: call 0x103f
19ee: mov r_00,0x0070
19f5: call 0x103f
19fa: mov r_00,0x0061
1a01: call 0x103f
1a06: mov r_00,0x0073
1a0d: call 0x103f
1a12: mov r_00,0x0073
1a19: call 0x103f
1a1e: mov r_00,0x0077
1a25: call 0x103f
1a2a: mov r_00,0x006f
1a31: call 0x103f
1a36: mov r_00,0x0072
1a3d: call 0x103f
1a42: mov r_00,0x0064
1a49: call 0x103f
1a4e: mov r_00,0x0021
1a55: call 0x103f
1a5a: mov r_00,0x000a
1a61: call 0x103f
1a66: pop r_0a
1a68: pop r_09
1a6a: ret
1a6d: mov r_00,0x0050
1a74: call 0x103f
1a79: mov r_00,0x006c
1a80: call 0x103f
1a85: mov r_00,0x0065
1a8c: call 0x103f
1a91: mov r_00,0x0061
1a98: call 0x103f
1a9d: mov r_00,0x0073
1aa4: call 0x103f
1aa9: mov r_00,0x0065
1ab0: call 0x103f
1ab5: mov r_00,0x0020
1abc: call 0x103f
1ac1: mov r_00,0x0065
1ac8: call 0x103f
1acd: mov r_00,0x006e
1ad4: call 0x103f
1ad9: mov r_00,0x0074
1ae0: call 0x103f
1ae5: mov r_00,0x0065
1aec: call 0x103f
1af1: mov r_00,0x0072
1af8: call 0x103f
1afd: mov r_00,0x0020
1b04: call 0x103f
1b09: mov r_00,0x0079
1b10: call 0x103f
```



```
1b15: mov r_00,0x006f
1b1c: call 0x103f
1b21: mov r_00,0x0075
1b28: call 0x103f
1b2d: mov r_00,0x0072
1b34: call 0x103f
1b39: mov r_00,0x0020
1b40: call 0x103f
1b45: mov r_00,0x0070
1b4c: call 0x103f
1b51: mov r_00,0x0061
1b58: call 0x103f
1b5d: mov r_00,0x0073
1b64: call 0x103f
1b69: mov r_00,0x0073
1b70: call 0x103f
1b75: mov r_00,0x0077
1b7c: call 0x103f
1b81: mov r_00,0x006f
1b88: call 0x103f
1b8d: mov r_00,0x0072
1b94: call 0x103f
1b99: mov r_00,0x0064
1ba0: call 0x103f
1ba5: mov r_00,0x003a
1bac: call 0x103f
1bb1: mov r_00,0x0020
1bb8: call 0x103f
1bbd: ret
1bc0: push r_09
1bc3: push r_0a
1bc6: push r_0b
1bc9: call 0x1a6d
1bce: mov r_01,0x001e
1bd5: mov r_00,0x0004
1bdc: call 0x1080
1be1: mov r_0b,r_00
1be5: jmp 0x1bef
1bea: jmp 0x1d7a
1bef: mov r_09,0x0000
1bf6: jmp 0x1d66
1bfb: mov r_1e,r_09
1bff: r_1e = r_1e * 0004
1c07: mov r_00,r_0b
1c0b: r_1e=r_1e + r_00
1c10: mov r_0a,r_1e
1c14: call 0x104b
1c19: mov r_01,r_00
1c20: mov [r_0a],r_01
1c20: mov r_01,[r_0a]
1c23: mov r_1e,r_01
1c27: mov r_00,0x000a
1c2e: cmp r_1e,r_00
1c32: jz 0x1c3c
1c37: jmp 0x1d5e
1c3c: mov r_00,0x0053
1c43: call 0x103f
1c48: mov r_00,0x006f
1c4f: call 0x103f
1c54: mov r_00,0x0072
1c5b: call 0x103f
1c60: mov r_00,0x0072
1c67: call 0x103f
1c6c: mov r_00,0x0079
1c73: call 0x103f
1c78: mov r_00,0x002c
1c7f: call 0x103f
1c84: mov r_00,0x0020
1c8b: call 0x103f
1c90: mov r_00,0x0077
1c97: call 0x103f
1c9c: mov r_00,0x0072
```

```
1ca3: call 0x103f
1ca8: mov r_00,0x006f
1caf: call 0x103f
1cb4: mov r_00,0x006e
1cbb: call 0x103f
1cc0: mov r_00,0x0067
1cc7: call 0x103f
1ccc: mov r_00,0x0020
1cd3: call 0x103f
1cd8: mov r_00,0x0070
1cdf: call 0x103f
1ce4: mov r_00,0x0061
1ceb: call 0x103f
1cf0: mov r_00,0x0073
1cf7: call 0x103f
1cfc: mov r_00,0x0073
1d03: call 0x103f
1d08: mov r_00,0x0077
1d0f: call 0x103f
1d14: mov r_00,0x006f
1d1b: call 0x103f
1d20: mov r_00,0x0072
1d27: call 0x103f
1d2c: mov r_00,0x0064
1d33: call 0x103f
1d38: mov r_00,0x0021
1d3f: call 0x103f
1d44: mov r_00,0x000a
1d4b: call 0x103f
1d50: mov r_00,0x0000
1d57: pop r_0b
1d59: pop r_0a
1d5b: pop r_09
1d5d: ret
1d5e: r_09=r_09+0001
1d66: mov r_1e,r_09
1d6a: mov r_00,0x001e
1d71: cmp r_1e,r_00
1d75: js 0x1bfb
1d7a: call 0x104b
1d7f: mov r_01,r_00
1d83: mov r_1e,r_01
1d87: mov r_00,0x000a
1d8e: cmp r_1e,r_00
1d92: jnz 0x1d9c
1d97: jmp 0x1ebe
1d9c: mov r_00,0x0053
1da3: call 0x103f
1da8: mov r_00,0x006f
1daf: call 0x103f
1db4: mov r_00,0x0072
1dbb: call 0x103f
1dc0: mov r_00,0x0072
1dc7: call 0x103f
1dcc: mov r_00,0x0079
1dd3: call 0x103f
1dd8: mov r_00,0x002c
1ddf: call 0x103f
1de4: mov r_00,0x0020
1deb: call 0x103f
1df0: mov r_00,0x0077
1df7: call 0x103f
1dfc: mov r_00,0x0072
1e03: call 0x103f
1e08: mov r_00,0x006f
1e0f: call 0x103f
1e14: mov r_00,0x006e
1e1b: call 0x103f
1e20: mov r_00,0x0067
1e27: call 0x103f
1e2c: mov r_00,0x0020
1e33: call 0x103f
```

```
1e38: mov r_00,0x0070
1e3f: call 0x103f
1e44: mov r_00,0x0061
1e4b: call 0x103f
1e50: mov r_00,0x0073
1e57: call 0x103f
1e5c: mov r_00,0x0073
1e63: call 0x103f
1e68: mov r_00,0x0077
1e6f: call 0x103f
1e74: mov r_00,0x006f
1e7b: call 0x103f
1e80: mov r_00,0x0072
1e87: call 0x103f
1e8c: mov r_00,0x0064
1e93: call 0x103f
1e98: mov r_00,0x0021
1e9f: call 0x103f
1ea4: mov r_00,0x000a
1eab: call 0x103f
1eb0: mov r_00,0x0000
1eb7: pop r_0b
1eb9: pop r_0a
1ebb: pop r_09
1ebd: ret
1ebe: mov r_00,r_0b
1ec2: call 0x12a9
1ec7: mov r_00,0x0000
1ece: pop r_0b
1ed0: pop r_0a
1ed2: pop r_09
1ed4: ret
```