Reversing a game script interpreter

Pierre Bourdon

July 8, 2011

Introduction

- Embedding a scripting language in software is really common
- Some people use well known languages (Lua, Python)
- Other people like to reinvent the world...

Common usage of scripting languages

- Games
- GUI (Python, QML, Javascript)
- Also used to obfuscate code

Plan

1. Interpreter architecture 101
    - Bytecode
    - Main loop

Plan

1. Interpreter architecture 101
   - Bytecode
   - Main loop

Script file compilation

- Scripts are usually made to be executed several times
- Parsing a language and analyzing code is slow
- It is a lot more efficient to compile the script to an interpreter specific bytecode which is then run when needed
- Bytecodes are made to be compact, fast to load and fast to execute

Stack based bytecode

```
PUSH ma_fonction
PUSH 6
PUSH 7
MUL
CALL
```

- Simple instructions, compact instruction set
- A lot of instructions must be executed to perform even simple tasks

Register based bytecode

```
MOV 6, R1
MOV 7, R2
MUL R1, R2, R3
CALL ma_fonction, R3
```

- Instructions are more complicated and take several operands
- Less instructions are needed

Plan

1. Interpreter architecture 101
   - Bytecode
   - Main loop

Main interpreter loop

- Set PC = entry point offset
- Execute the instruction at PC and increment PC
- Repeat until an EXIT instruction is reached

Executing an instruction

- Take the instruction opcode and lookup in a table the code to execute for this opcode
- There are more complex methods (direct threading, indirect threading) which are faster but more difficult to implement
- Most of the interpreter code is in the instruction handlers

Plan

2 Reversing CScript

- Finding the interpreter code
- Dumping memory accesses
- Instruction dispatcher
- Categorizing data accesses

What is CScript?

- From a Wii RPG: *Tales of Symphonia 2*
- Used to control characters and animations during cinematic scenes
- Also used to script game events
- Only used in this game as far as I know

Plan

2 Reversing CScript

- Finding the interpreter code
- Dumping memory accesses
- Instruction dispatcher
- Categorizing data accesses

Method

- Find the address where the bytecode is in memory
- Use it to find the code which access the bytecode in memory
- We can safely assume that it is the interpreter

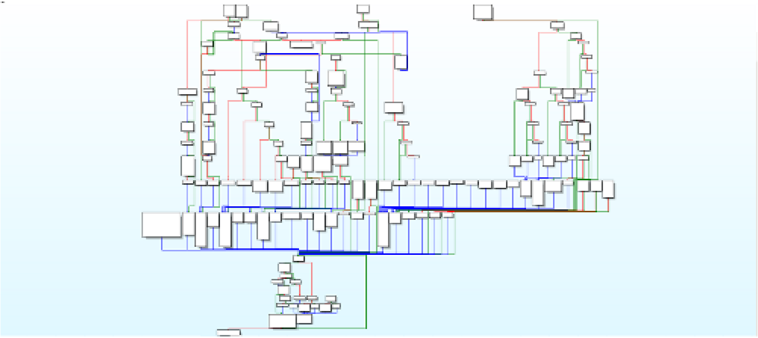
Finding the bytecode in memory

- Run the program in a debugger
- During the script execution, freeze the process
- Dump the process memory
- Search the bytecode in it using grep -b/hexedit/whatever

Finding code reading the bytecode

- If you’ve got a correct debugger which can place memory breakpoints, it’s easy
- If you don’t, run the code in an emulator and modify the memory load code to log the instruction offset

Main interpreter loop



- If we look carefully, we can see some blocks of code without direct predecessors
- This often means dispatch table, which in this case is used to dispatch instructions

Plan

2 Reversing CScript

- Finding the interpreter code
- Dumping memory accesses
- Instruction dispatcher
- Categorizing data accesses

Why?

- Understanding the code is hard
- It's easier to think about data than code
- We know more or less what to expect in the interpreter state

How?

- If you are executing the interpreter through an emulator, simply modify the emulator code
- If you don't but your debugger supports memory breakpoints, use this
- Dump the whole CPU state and the memory access type (read/write, size)

Example

```
{’type’:’r’, ’size’:4, ’addr’:’016E00F4’,
 ’val’: ’11000000’, ’pc’:’80091DBC’,
 ’r0’:’00000000’, ’r1’:’807AB378’, ...}
```

- Log to an easily parsable format!

Plan

2 Reversing CScript
- Finding the interpreter code
- Dumping memory accesses
- Instruction dispatcher
- Categorizing data accesses

Finding the instruction dispatcher

- Dispatching works by loading the opcode then doing an indirect jump to an address in a table
- If you're a wizard, find the instruction loading the opcode by reading the ASM
- If you're not, use the memory dump to find the most executed instruction which reads the bytecode

Deducing where PC is stored

- To load the current opcode, we need to be able to define "current"
- The interpreter keeps the current offset in its state
- The instruction dispatcher needs to read this offset to load the opcode

Example

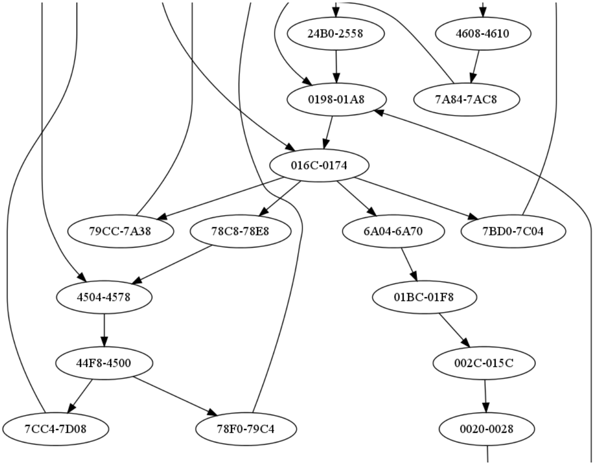
```
lwz     %r0, 0xC(%r15)
lwz     %r4, 4(%r15)
mulli   %r0, %r0, 0x1420
add     %r5, %r15, %r0
lwz     %r3, 0x142C(%r5)
lwzx    %r17, %r4, %r3
```

- `r15` contains the interpreter state
- The bytecode address is loaded into `r4`
- The current PC is loaded into `r3`
- The last instruction loads the opcode

Somewhat useful application

- We have the list of offsets where the executed opcodes are
- When there is a gap between two consecutive offsets we can assume it's a jump or a call
- Let's look at the script control flow!

Script control flow



- This is mostly useless but a nice proof of concept :)
- *Mostly* useless

Finding control flow opcodes

- We can look the opcodes which trigger a control flow change
- JMP, CALL, RET
- Conditional jump
- That's already 4 instructions easily reversed

Plan

2 Reversing CScript

- Finding the interpreter code
- Dumping memory accesses
- Instruction dispatcher
- Categorizing data accesses

Finding the stack

- CALL and RET store addresses in the stack
- This can be found in the memory access logs
- If there is a stack it is likely to be used for things like argument passing or local variables

Finding the eventual registers

- The easiest way is to search for an instruction doing things like floating point division
- There are very few chances to find that outside of variables handling
- We can then find from where are our variables loaded

Categorizing data accesses

- Through a lot of work you'll begin how the interpreter state is stored
- With these infos you can make our data accesses log more useful

Example

```
{’type’:’r’, ’size’:4, ’addr’:’016E00F4’,
 ’val’: ’11000000’, ’pc’:’80091DBC’,
 ’r0’:’00000000’, ’r1’:’807AB378’, ...}
```

```
ReadInstr: 11000000 at pc=00007D40 (@ 80091DBC)
```

Another example

```
{’type’:’r’, ’size’:4, ’addr’:’016E1964’,
’val’: ’00007D40’, ’pc’:’80091DB8’,
’r0’:’00000000’, ’r1’:’807AB378’, ...}
```

```
GetPC: 00007D40 at addr=016E1964 (@ 80091DB8)
```

Reversing simple instructions from dumps

Some instructions of the bytecode are really simple to reverse when you have a readable memory dump of their execution

```
ReadInstr: 08000000 at pc=6E24
SetPC: 6E28
GetArg: 00006D80 at pc=6E28
SetPC: 6E2C
SetPC: 6D80
```

Sadly...

- There is not always enough informations to understand an instruction from its memory accesses dump
- Instructions which are used a lot may can be reversed by comparing the input values (regs, stack) to their behavior
- Reading the assembly is always needed to be sure to not miss things!

Plan

3 Conclusion

Conclusion

- Reversing an interpreter is hard and takes time
- There is no generic method to do all the work
- However there are methods to make analysis easier

Questions?

- http://blog.delroth.net/
- http://code.delroth.net/cscript-interpreter/
- @delroth_