Bio

About Me

Almost Every Weekend

With VN Security since year 2009

- CTF player
- Weekend gamer

Most of the time

Running zxandora.com project.

- Soon
- Very Soon
- Brand New Online Sandbox

Once a year

Hack in The Box Crew

- Good friends
- CTF CTF and CTF

About Me

- 2008, Hack In The Box CTF Winner
- 2010, Hack In The Box Speaker, Malaysia
- 2012, Codegate Speaker, Korea
- 2015, VXRL Speaker, Hong Kong
- 2015, HITCON CTF, Prequal Top 10
- 2016, Codegate CTF, Prequal Top 5
- 2016, Qcon Speaker, Beijing

- OSX, Local Privilege Escalation
- Code commit for metasploit 3
- GDB Bug hunting
- Metasploit module
- Linux Randomization Bypass
- http://www.githiub.com/xwings/tuya
- 微博: @kaijern

vnsecurity.net

Introduction

VN Security

- Active CTF Player (CLGT)
- Active speaker at conferences
  - Blackhat USA
  - Tetcon
  - Hack In The Box
  - Xcon
- Our Tools
  - PEDA
  - Unicorn/ Capstone/ Keystone
  - Xandora
  - OllyDbg, Catcha!
  - ROPEME
- Nations
  - Vietnamese
  - Malaysian
  - Singaporean

Nguyen Anh Quynh

- Security Researcher
- Active speaker at conferences
  - Blackhat USA
  - Syscan
  - Hack In The Box
  - Xcon
- Research Topics
  - Emulators
  - Virtualization
  - Binary Analysis
  - Tools for Malware Analysis

When gdb meets peda

GDB

PEDA

```
(gdb) disassemble
Dump of assembler code for function main:
0x000000000040058c <main+0>:    push   %rbp
0x000000000040058d <main+1>:    mov    %rsp,%rbp
0x0000000000400590 <main+4>:    sub    $0x10,%rsp
0x0000000000400594 <main+8>:    mov    $0x4,%edi
0x0000000000400599 <main+13>:   callq  0x4004a8 <_init+56>
0x000000000040059e <main+18>:   mov    %rax,0xfffffffffffffff0(%rbp)
0x00000000004005a2 <main+22>:   movl   $0x0,0xfffffffffffffffc(%rbp)
0x00000000004005a9 <main+29>:   mov    0xfffffffffffffffc(%rbp),%eax
0x00000000004005ac <main+32>:   cltq
0x00000000004005ae <main+34>:   shl    $0x2,%rax
0x00000000004005b2 <main+38>:   mov    %rax,%rdx
0x00000000004005b5 <main+41>:   add    0xfffffffffffffff0(%rbp),%rdx
0x00000000004005b9 <main+45>:   mov    0xfffffffffffffffc(%rbp),%eax
0x00000000004005bc <main+48>:   mov    %eax,(%rdx)
0x00000000004005be <main+50>:   mov    0xfffffffffffffffc(%rbp),%eax
0x00000000004005c1 <main+53>:   cltq
0x00000000004005c3 <main+55>:   shl    $0x2,%rax
0x00000000004005c7 <main+59>:   add    0xfffffffffffffff0(%rbp),%rax
0x00000000004005cb <main+63>:   mov    (%rax),%edx
0x00000000004005cd <main+65>:   mov    0xfffffffffffffffc(%rbp),%esi
0x00000000004005d0 <main+68>:   mov    $0x4006dc,%edi
0x00000000004005d5 <main+73>:   mov    $0x0,%eax
0x00000000004005da <main+78>:   callq  0x4004b8 <_init+72>
0x00000000004005df <main+83>:   addl   $0x1,0xfffffffffffffffc(%rbp)
0x00000000004005e3 <main+87>:   jmp    0x4005a9 <main+29>
End of assembler dump.
(gdb)
```

```
gdb-peda$ start
[----------------------------------registers-----------------------------------]
EAX: 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
EBX: 0xb7fcbff4 --> 0x155d7c
ECX: 0xd5eeaa03
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xbffff748 --> 0xbffff7c8 --> 0x0
ESP: 0xbffff748 --> 0xbffff7c8 --> 0x0
EIP: 0x80483e7 (<main+3>:       and    esp,0xfffffff0)
EFLAGS: 0x200246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80483e3 <frame_dummy+35>:  nop
   0x80483e4 <main>:    push   ebp
   0x80483e5 <main+1>:  mov    ebp,esp
=> 0x80483e7 <main+3>:  and    esp,0xfffffff0
   0x80483ea <main+6>:  sub    esp,0x110
   0x80483f0 <main+12>: mov    eax,DWORD PTR [ebp+0xc]
   0x80483f3 <main+15>: add    eax,0x4
   0x80483f6 <main+18>: mov    eax,DWORD PTR [eax]
[------------------------------------stack-------------------------------------]
0000| 0xbffff748 --> 0xbffff7c8 --> 0x0
0004| 0xbffff74c --> 0xb7e8cbd6 (<__libc_start_main+230>:       mov    DWORD PTR [e
0008| 0xbffff750 --> 0x1
0012| 0xbffff754 --> 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
0016| 0xbffff758 --> 0xbffff7fc --> 0xbffff922 ("SHELL=/bin/bash")
0020| 0xbffff75c --> 0xb7fe1858 --> 0xb7e76000 --> 0x464c457f
0024| 0xbffff760 --> 0xbffff7b0 --> 0x0
0028| 0xbffff764 --> 0xffffffff
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Temporary breakpoint 1, 0x080483e7 in main ()
gdb-peda$
```

Why KCON

Fake Websites

What Are These Things

What Is Disassembler

- From binary to assembly code
- Core part of all binary analysis/ reverse engineering / debugger and exploit development
- Disassembly framework (engine/library) is a lower layer in stack of architecture

Example

- 01D8 = ADD EAX,EBX (x86)
- 1169 = STR R1,[R2] (ARM's Thumb)

Binary Analysis | Debugger | Exploit Development

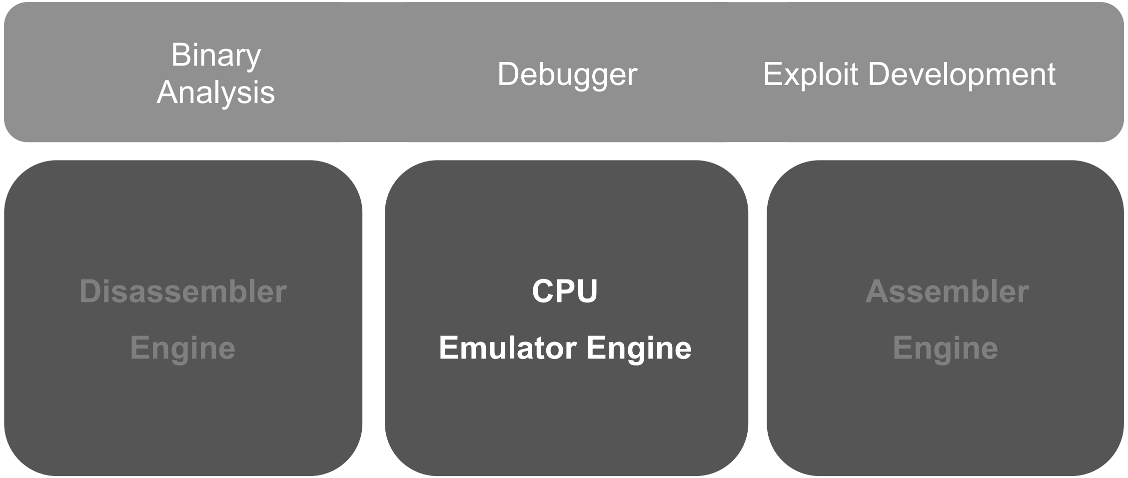
Disassembler Engine | **CPU Emulator Engine** | **Assembler Engine**

What Is Emulator

- Software only CPU Emulator
- Core focus on CPU operations.
- Design with no machine devices
- Safe emulation environment
- Where else can we see CPU emulator. Yes, Antivirus

Example

- 01D1 = add eax,ebx (x86)
  - Load eax & ebx register
  - Add value of eax & ebx then copy the result to eax
  - Update flag OF, SF, ZF, AF, CF, PF accordingly



What Is Assembler

- From assembly to machine code
- Support high level concepts such as macro, functions and etc.
- Dynamic machine code generation

Example

- ADD EAX,EBX = 01D8 (x86)
- STR R1,[R2] = 1169 (ARM's Thumb)

Binary Analysis | Debugger | Exploit Development

Disassembler Engine | CPU Emulator Engine | **Assembler Engine**

Where are we currently

Showcase

- CEnigma
- Unicorn
- CEbot
- Camal
- Radare2
- Pyew
- WinAppDbg
- PowerSploit
- MachOview
- RopShell
- ROPgadget
- Frida
- The-Backdoor-Factory
- Cuckoo
- Cerbero Profiler
- CryptoShark
- Ropper
- Snowman
- X86dbg
- Concolica
- Memtools Vita
- BARF
- rp++
- Binwalk
- MPRESS dumper
- Xipiter Toolkit
- Sonare
- PyDA
- Qira
- Rekall
- Inficere
- Pwntools
- Bokken
- Webkitties
- Malware_config_parsers
- Nightmare
- Catfish
- JSoS-Module-Dump
- Vitasploit
- PowerShellArsenal
- PyReil
- ARMSCGen
- Shwass
- Nrop
- Illdb-capstone-arm
- Capstone-js
- ELF Unstrip Tool
- Binjitsu
- Rop-tool
- JitAsm
- OllyCapstone
- PackerId
- Volatility Plugins
- Pwndbg
- Lisa.py
- Many Other More

Showcase

- UniDOS: Microsoft DOS emulator.
- Radare2: Unix-like reverse engineering framework and commandline tools.
- Usercorn: User-space system emulator.
- Unicorn-decoder: A shellcode decoder that can dump self-modifying-code.
- Univm: A plugin for x64dbg for x86 emulation.
- PyAna: Analyzing Windows shellcode.
- GEF: GDB Enhanced Features.
- Pwndbg: A Python plugin of GDB to assist exploit development.
- Eli.Decode: Decode obfuscated shellcodes.
- IdaEmu: an IDA Pro Plugin for code emulation.
- Roper: build ROP-chain attacks on a target binary using genetic algorithms.
- Sk3wlDbg: A plugin for IDA Pro for machine code emulation.
- Angr: A framework for static & dynamic concolic (symbolic) analysis.
- Cemu: Cheap EMUlator based on Keystone and Unicorn engines.
- ROPMEMU: Analyze ROP-based exploitation.
- BroIDS_Unicorn: Plugin to detect shellcode on Bro IDS with Unicorn.
- UniAna: Analysis PE file or Shellcode (Only Windows x86).
- ARMSCGen: ARM Shellcode Generator.
- TinyAntivirus: Open source Antivirus engine designed for detecting & disinfecting polymorphic virus.
- Patchkit: A powerful binary patching toolkit.

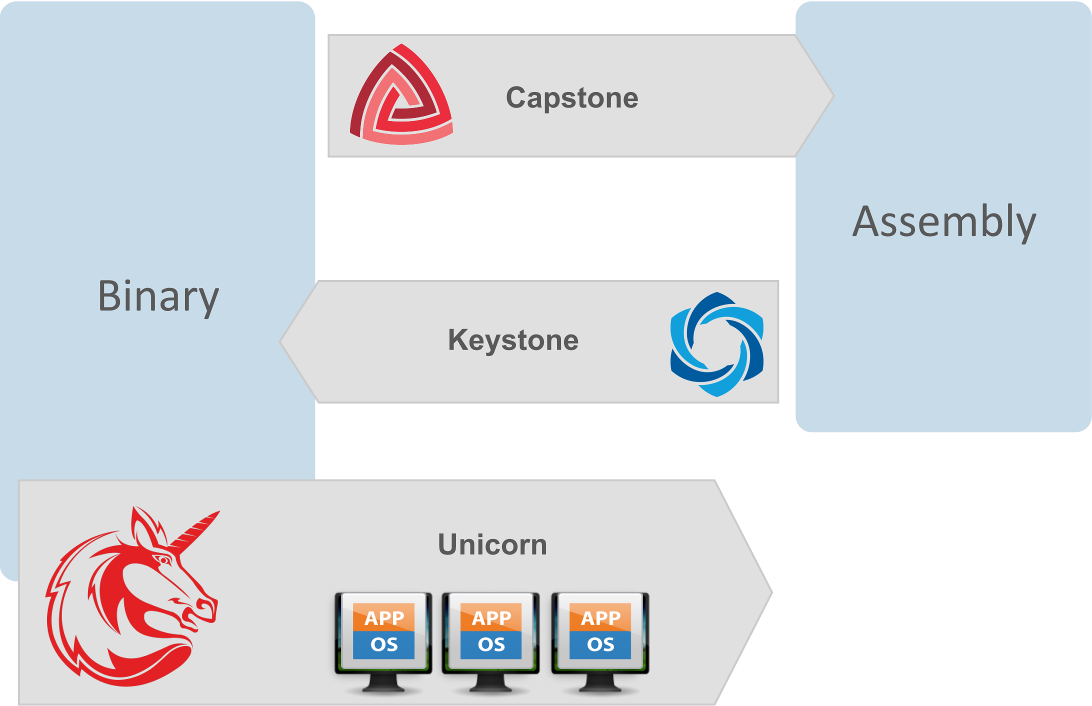
Showcase

- Keypatch: IDA Pro plugin for code assembling & binary patching.
- Radare2: Unix-like reverse engineering framework and commandline tools.
- GEF: GDB Enhanced Features.
- Ropper: Rop gadget and binary information tool.
- Cemu: Cheap EMUlator based on Keystone and Unicorn engines.
- Pwnypack: Certified Edible Dinosaurs official CTF toolkit.
- Keystone.JS: Emscripten-port of Keystone for JavaScript.
- Usercorn: Versatile kernel+system+userspace emulator.
- x64dbg: An open-source x64/x32 debugger for windows.
- Liberation: a next generation code injection library for iOS cheaters everywhere.
- Strongdb: GDB plugin for Android debugging.
- AssemblyBot: Telegram bot for assembling and disassembling on-the-go.
- demovfuscator: Deobfuscator for movfuscated binaries.
- Dash: A simple web based tool for working with assembly language.
- ARMSCGen: ARM Shellcode Generator.
- Asm_Ops: Assembler for IDA Pro (IDA Plugin).
- Binch: A lightweight ELF binary patch tool.
- Metame: Metamorphic code engine for arbitrary executables.
- Patchkit: A powerful binary patching toolkit.
- Pymetamorph: Metamorphic engine in Python for Windows executables.

Born of The Trinity

Fundamental Frameworks for Reversing

- Components for a complete RE framework
- Interchange between assembler and disassembler
- A full CPU emulator always help when comes with obfuscated code



Capstone Engine

NGUYEN Anh Quynh <aquynh -at- gmail.com>

http://www.capstone-engine.org

What's Wrong with Current Disassembler

Features	Distorm3	BeaEngine	Udis86	Libopcode
X86|Arm	✓|X	✓|X	✓|X	✓|✓ [1]
Linux|Windows	✓|✓	✓|✓	✓|✓	✓|X
Python|Ruby bindings	✓|X [2]	✓|X	✓|X	✓|X
Update	X	?	X	X
License	GPL	LGPL3	BSD	GPL

- Nothing works even up until 2013 (First release of Capstone Engine)
- Looks like no one take charge
- Industry stays in the dark side

What do we need ?

- Multiple archs: x86, ARM+ ARM64 + Mips + PPC and more
- Multiple platform: Windows, Linux, OSX and more
- Multiple binding: Python, Ruby, Java, C# and more

- Clean, simple, intuitive & architecture-neutral API
- Provide break-down details on instructions
- Friendly license: Not GPL

Lots of Work !

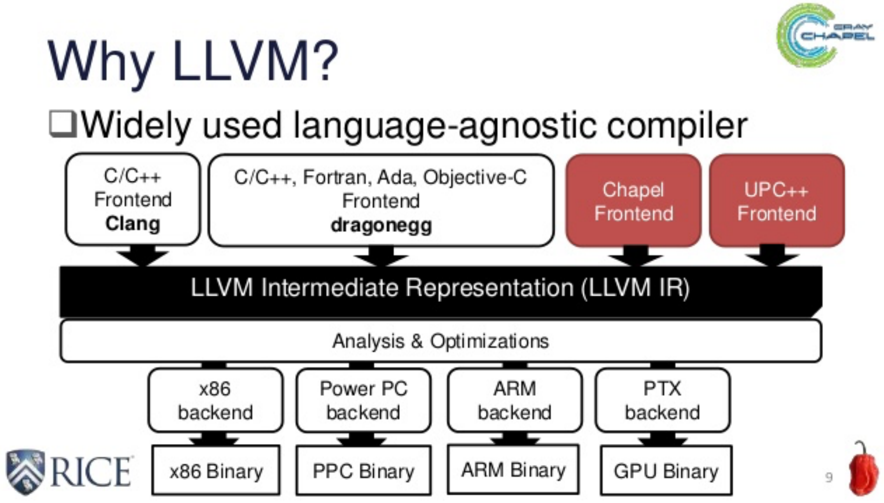
- Multiple archs: x86, ARM
- Actively maintained & update within latest arch's change
- Multiple platform: Windows, Linux
- Understanding opcode, Intel x86 it self with 1500++ documented instructions

- Support python and ruby as binding languages
- Single man show
- Target finish within 12 months

A Good Disassembler

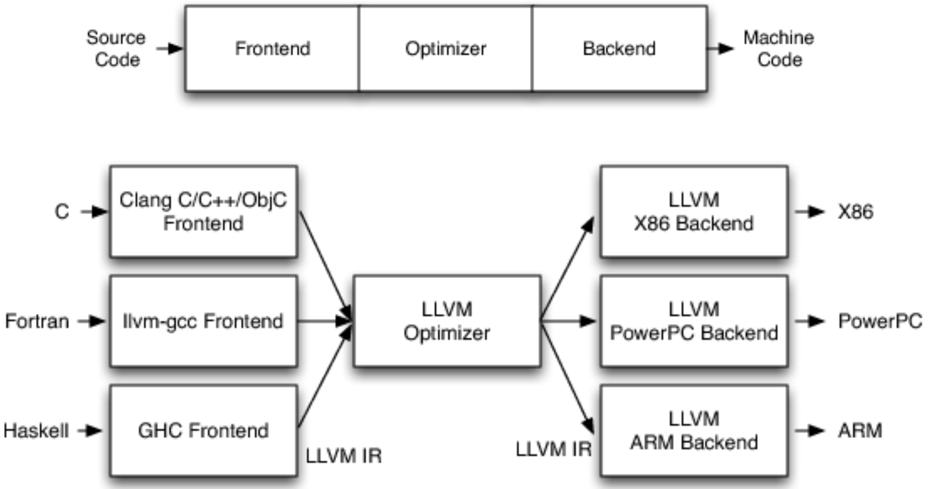
- Multiple archs: x86, ARM
- Actively maintained & update within latest arch's change
- Multiple platform: Windows, Linux
- Support python and ruby as binding languages
- Friendly license: BSD
- Easy to setup

Not Reinventing the Wheel



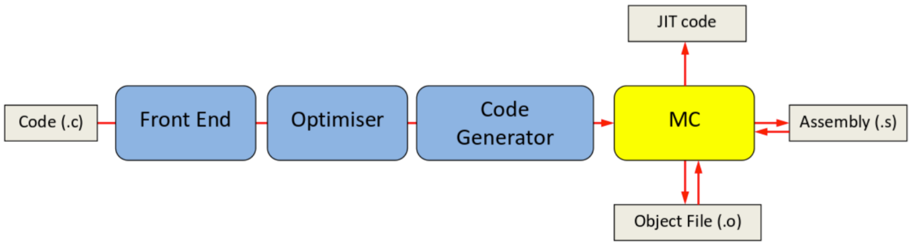
- Open source project compiler
- Sets of modules for machine code representing, compiling, optimizing
- Backed by many major players: AMD, Apple, Google, Intel, IBM, ARM, Imgtec, Nvidia, Qualcomm, Samsung, etc
- Incredibly huge (compiler) community around.

Fork from LLVM



- Multiple architectures ready
- In-disassembler (MC module)
  - Only, Only and Only build for LLVM
  - actively maintained by the original vendor from the arch building company (eg, x86 from intel)
- Very actively maintained & updated by a huge community

Are We Done



Issues

- Cannot just reuse MC as-is without huge efforts.
  - LLVM code is in C++, but we want C code.
  - Code mixed like spaghetti with lots of LLVM layers, not easy to take out
  - Need to build instruction breakdown-details ourselves.
  - Expose semantics to the API.
  - Not designed to be thread-safe.
  - Poor Windows support.
- Need to build all bindings ourselves.
- Keep up with upstream code once forking LLVM to maintain ourselves.

Solutions

- Fork LLVM but must remove everything we do not need
- Replicated LLVM's MC
  - Build around MC and not changing MC
  - Replace C++ with C
- Extend LLVM's MC
  - Isolate some global variable to make sure thread-safe
- Semantics information from TD file from LLVM
- cs_inn structure
  - Keep all information and group nicely
  - Make sure API are arch-independent

Capstone is not LLVM

More Superiors

- Zero dependency
- Compact in size
- More than assembly code
- Thread-safe design
- Able to embed into restricted firmware OS/ Environments
- Malware resistance (x86)
- Optimized for reverse engineers
- More hardware mode supported:- Big-Endian for ARM and ARM64
- More Instructions supported: 3DNow (x86)

More Robust

- Cannot always rely on LLVM to fix bugs
    - Disassembler is still conferred seconds-class LLVM, especially if does not affect code generation
    - May refuse to fix bugs if LLVM backed does not generate them (tricky x86 code)
- But handle all comer case properly is Capstone first priority
    - Handle all x86 malware ticks we aware of
    - LLVM could not care less

Demo

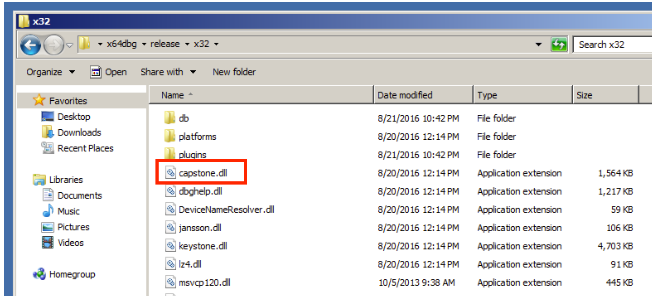
```c
/* test1.c */

#include <stdio.h>
#include <inttypes.h>

#include <capstone/capstone.h>

#define CODE "\x55\x48\x8b\x05\xb8\x13\x00\x00"

int main(void)
{
  csh handle;
  cs_insn *insn;
  size_t count;

  if (cs_open(CS_ARCH_X86, CS_MODE_64, &handle) != CS_ERR_OK)
      return -1;
  count = cs_disasm(handle, CODE, sizeof(CODE)-1, 0x1000, 0, &insn);
  if (count > 0) {
      size_t j;
      for (j = 0; j < count; j++) {
          printf("0x%"PRIx64":\t%s\t\t%s\n", insn[j].address, insn[j].mnemoni
                  insn[j].op_str);
      }

      cs_free(insn, count);
  } else
      printf("ERROR: Failed to disassemble given code!\n");

  cs_close(&handle);

     return 0;
}
```

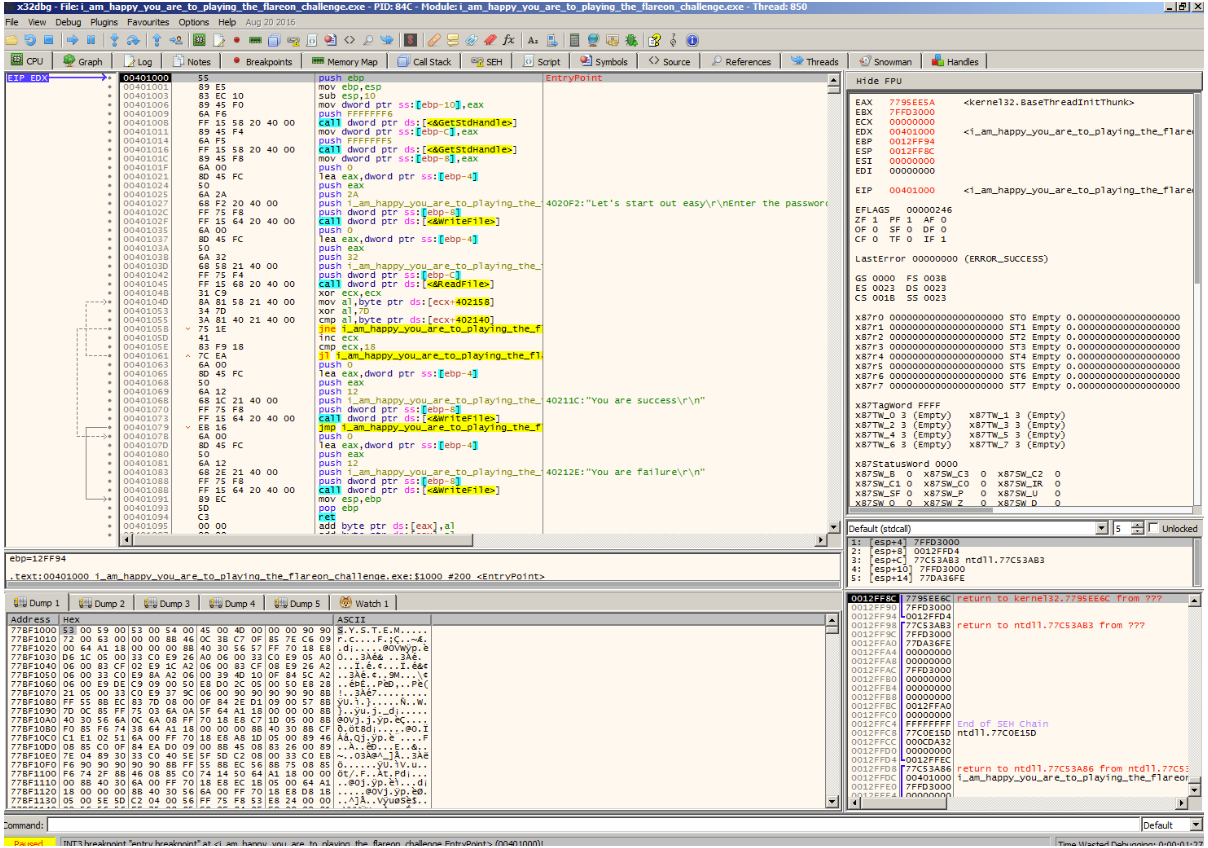
```python
# test1.py
from capstone import *

CODE = b"\x55\x48\x8b\x05\xb8\x13\x00\x00"

md = Cs(CS_ARCH_X86, CS_MODE_64)
for i in md.disasm(CODE, 0x1000):
    print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
```

```
$ make
cc -c test1.c -o test1.o
cc test1.o -O3 -Wall -lcapstone -o test1

$ ./test1
0x1000: push        rbp
0x1001: mov     rax, qword ptr [rip + 0x13b8]
```

```
$ python test1.py

0x1000: push    rbp
0x1001: mov rax, qword ptr [rip + 0x13b8]
```

Showcase: x64dbg





Unicorn Engine

NGUYEN Anh Quynh <aquynh -at- gmail.com>
DANG Hoang Vu <danghvu -at- gmail.com>

http://www.unicorn-engine.org

What's Wrong with Current Emulator

Features	libemu	PyEmu	IDA-x86emu	libCPU
Multi-arch	X	X	X	X [1]
Updated	X	X	X	X
Independent	X [2]	X [3]	X [4]	✓
JIT	X	X	X	✓

- Nothing works even up until 2015 (First release of Unicorn Engine)
- Limited bindings
- Limited functions, limited architecture

What Do We Need ?

Features	libemu	PyEmu	IDA-x86emu	libCPU	Unicorn
Multi-arch	X	X	X	X	✓
Updated	X	X	X	X	✓
Independent	X	X	X	✓	✓
JIT	X	X	X	✓	✓

- Multiple archs: x86, x86_64, ARM+ ARM64 + Mips + PPC
- Multiple platform: Windows, Linux, OSX, Android and more
- Multiple binding: Python, Ruby, Java, C# and more

- Pure C implementation
- Latest and updated architecture
- With JIT compiler technique
- Instrumentation eg. F7, F8

Lots of Work !

- Multiple archs: x86, ARM
- Actively maintained & update within latest arch's change
- Multiple platform: Windows, Linux
- Understanding opcode, Intel x86 it self with 1500++ documented instructions

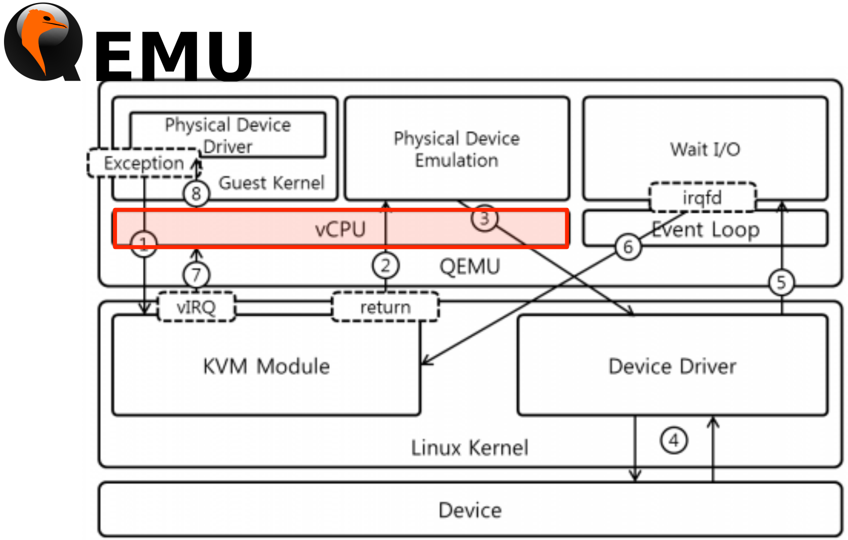
- Support python and ruby as binding languages
- Single man show
- Target finish within 12 months

A Good Emulator

- Multiple archs: x86, x86_64, ARM, ARM64, Mips and more
- Actively maintained & update within latest arch’s change
- Multiple platform: Windows, Linux, OSX, Android and more

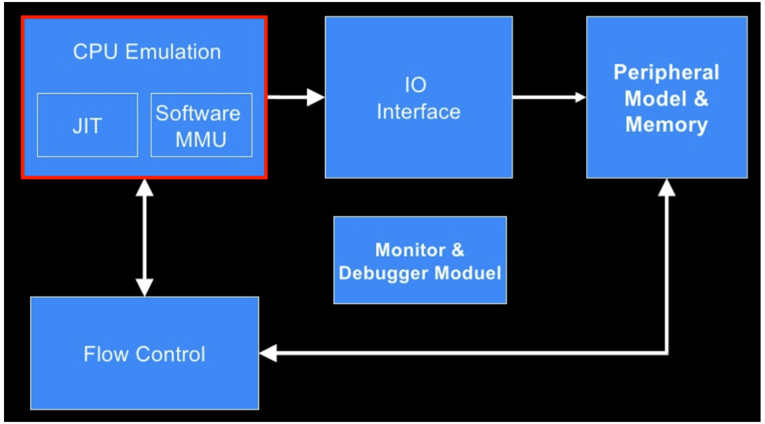
- Code in pure C
- Support python and ruby as binding languages
- JIT compiler technique
- Instrumentation at various level
  - Single step
  - Instruction
  - Memory Access

Not Reinventing the Wheel



- Open source project on system emulator
- Very huge community and highly active
- Multiple architecture: x86, ARM, ARM64, Mips, PowerPC, Sparc, etc (18 architectures)
- Multiple platform: *nix and Windows

Fork from QEMU



- Support all kind of architectures and very updated
- Already implemented in pure C, so easy to implement Unicorn core on top
- Already supported JIT in CPU emulation, optimization on of of JIT
- Are we done ?

Are We Done

Issues 1

- Not just emulate CPU, but also device models & ROM/BIOS to fully emulate physical machines
- Qemu codebase is huge and mixed like spaghetti
- Difficult to read, as contributed by many different people

Solutions

- Keep only CPU emulation code & remove everything else (devices, ROM/BIOS, migration, etc)
- Keep supported subsystems like Qobject, Qom
- Rewrites some components but keep CPU emulation code intact (so easy to sync with Qemu in future)

Issues 2

- Set of emulators for individual architecture
  - Independently built at compile time
  - All archs code share a lot of internal data structures and global variables
- Unicorn wants a single emulator that supports all archs

Solutions

- Isolated common variables & structures
  - Ensured thread-safe by design
- Refactored to allow multiple instances of Unicorn at the same time Modified the build system to support multiple archs on demand

Are We Done

Issues 3

- Instrumentation for static compilation only
- JIT optimizes for performance with lots of fast-path tricks, making code instrumenting extremely hard

Solutions

- Build dynamic fine-grained instrumentation layer from scratch Support various levels of instrumentation
  - Single-step or on particular instruction (TCG level)
  - Instrumentation of memory accesses (TLB level)
  - Dynamically read and write register
  - Handle exception, interrupt, syscall (arch-level) through user provided callback.

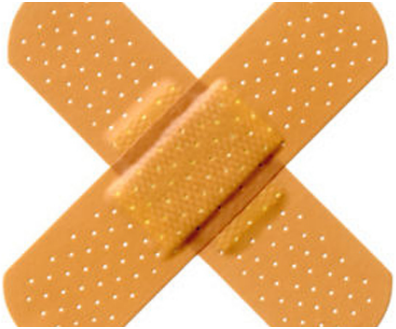
Issues 4

- Objects is open (malloc) without closing (freeing) properly everywhere
- Fine for a tool, but unacceptable for a framework

Solutions

- Find and fix all the memory leak issues
- Refactor various subsystems to keep track and cleanup dangling pointers

Unicorn Engine is not QEMU



- Independent framework
- Much more compact in size, lightweight in memory
- Thread-safe with multiple architectures supported in a single binary Provide interface for dynamic instrumentation
- More resistant to exploitation (more secure)
  - CPU emulation component is never exploited!
  - Easy to test and fuzz as an API.

Demo

```c
#include <unicorn/unicorn.h>

// code to be emulated
#define X86_CODE32 "\x41\x4a" // INC ecx; DEC edx

// memory address where emulation starts
#define ADDRESS 0x1000000

int main(int argc, char **argv, char **envp)
{
  uc_engine *uc;
  uc_err err;
  int r_ecx = 0x1234;     // ECX register
  int r_edx = 0x7890;     // EDX register

  printf("Emulate i386 code\n");

  // Initialize emulator in X86-32bit mode
  err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
  if (err != UC_ERR_OK) {
    printf("Failed on uc_open() with error returned: %u\n", err);
    return -1;
  }

  // map 2MB memory for this emulation
  uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);

  // write machine code to be emulated to memory
  if (uc_mem_write(uc, ADDRESS, X86_CODE32, sizeof(X86_CODE32) - 1)) {
    printf("Failed to write emulation code to memory, quit!\n");
    return -1;
  }

  // initialize machine registers
  uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
  uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);

  // emulate code in infinite time & unlimited instructions
  err=uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32) - 1, 0, 0
  if (err) {
    printf("Failed on uc_emu_start() with error returned %u: %s\n",
      err, uc_strerror(err));
  }

  // now print out some registers
  printf("Emulation done. Below is the CPU context\n");

  uc_reg_read(uc, UC_X86_REG_ECX, &r_ecx);
  uc_reg_read(uc, UC_X86_REG_EDX, &r_edx);
  printf(">>> ECX = 0x%x\n", r_ecx);
  printf(">>> EDX = 0x%x\n", r_edx);

  uc_close(uc);

  return 0;
}
```

```
$ make
cc  test1.c -L/usr/local/Cellar/glib/2.44.1/lib -L/usr/local/opt/gettext/l

$ ./test1
Emulate i386 code
Emulation done. Below is the CPU context
>>> ECX = 0x1235
>>> EDX = 0x788f
```

```python
from __future__ import print_function
from unicorn import *
from unicorn.x86_const import *

# code to be emulated
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx

# memory address where emulation starts
ADDRESS = 0x1000000

print("Emulate i386 code")
try:
    # Initialize emulator in X86-32bit mode
    mu = Uc(UC_ARCH_X86, UC_MODE_32)

    # map 2MB memory for this emulation
    mu.mem_map(ADDRESS, 2 * 1024 * 1024)

    # write machine code to be emulated to memory
    mu.mem_write(ADDRESS, X86_CODE32)

    # initialize machine registers
    mu.reg_write(UC_X86_REG_ECX, 0x1234)
    mu.reg_write(UC_X86_REG_EDX, 0x7890)

    # emulate code in infinite time & unlimited instructions
    mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))

    # now print out some registers
    print("Emulation done. Below is the CPU context")

    r_ecx = mu.reg_read(UC_X86_REG_ECX)
    r_edx = mu.reg_read(UC_X86_REG_EDX)
    print(">>> ECX = 0x%x" %r_ecx)
    print(">>> EDX = 0x%x" %r_edx)

except UcError as e:
    print("ERROR: %s" % e)
```

```
$ python test1.py

Emulate i386 code
Emulation done. Below is the CPU context
>>> ECX = 0x1235
>>> EDX = 0x788f
```

Showcase: box.py

```
(20:54:08):xwings@kali32:<~/box>
(4)$ hexdump -C samples/UrlDownloadToFile.sc
00000000  50 90 50 90 50 90 50 90  90 90 90 90 90 90 90 90  |P.P.P.P.........|
00000010  e9 fb 00 00 00 5f 64 a1  30 00 00 00 8b 40 0c 8b  |....._d.0....@..|
00000020  70 1c ad 8b 68 20 80 7d  0c 33 74 03 96 eb f3 8b  |p...h .}.3t.....|
00000030  68 08 8b f7 6a 04 59 e8  8f 00 00 00 e2 f9 68 6f  |h...j.Y.......ho|
00000040  6e 00 00 68 75 72 6c 6d  54 ff 16 8b e8 e8 79 00  |n..hurlmT.....y.|
00000050  00 00 8b d7 47 80 3f 00  75 fa 47 57 47 80 3f 00  |....G.?.u.GWG.?.|
00000060  75 fa 8b ef 5f 33 c9 81  ec 04 01 00 00 8b dc 51  |u..._3.........Q|
00000070  52 53 68 04 01 00 00 ff  56 0c 5a 59 51 52 8b 02  |RSh.....V.ZYQR..|
00000080  53 43 80 3b 00 75 fa 81  7b fc 2e 65 78 65 75 03  |SC.;.u..{..exeu.|
00000090  83 eb 08 89 03 c7 43 04  2e 65 78 65 c6 43 08 00  |......C..exe.C..|
000000a0  5b 8a c1 04 30 88 45 00  33 c0 50 50 53 57 50 ff  |[...0.E.3.PPSWP.|
000000b0  56 10 83 f8 00 75 06 6a  01 53 ff 56 04 5a 59 83  |V....u.j.S.V.ZY.|
000000c0  c2 04 41 80 3a 00 75 b4  ff 56 08 51 56 8b 75 3c  |..A.:.u..V.QV.u<|
000000d0  8b 74 35 78 03 f5 56 8b  76 20 03 f5 33 c9 49 41  |.t5x..V.v ..3.IA|
000000e0  ad 03 c5 33 db 0f be 10  38 f2 74 08 c1 cb 0d 03  |...3....8.t.....|
000000f0  da 40 eb f1 3b 1f 75 e7  5e 8b 5e 24 03 dd 66 8b  |.@..;.u.^.^$..f.|
00000100  0c 4b 8b 5e 1c 03 dd 8b  04 8b 03 c5 ab 5e 59 c3  |.K.^.........^Y.|
00000110  e8 00 ff ff ff 8e 4e 0e  ec 98 fe 8a 0e 7e d8 e2  |......N......~..|
00000120  73 33 ca 8a 5b 36 1a 2f  70 64 45 62 57 00 68 74  |s3..[6./pdEbW.ht|
00000130  74 70 3a 2f 2f 62 6c 61  68 62 6c 61 68 2e 63 6f  |tp://blahblah.co|
00000140  6d 2f 65 76 69 6c 2e 65  78 65 00 00 00 00 00 00  |m/evil.exe......|
00000150
```

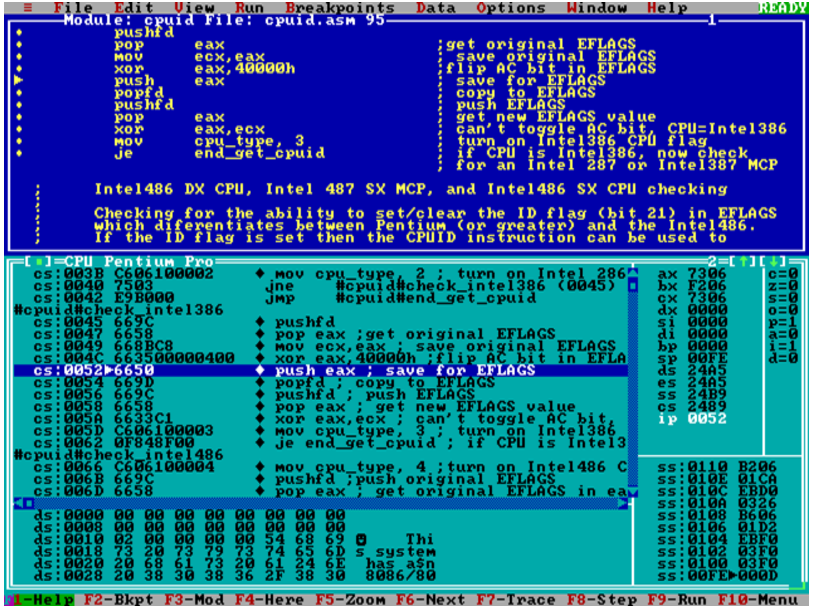
```python
def read_shellcode(fname):
    # get shellcode for emulation
    f = open(fname, 'rb')
    shellcode = f.read()
    f.close()
    return shellcode


# using Capstone for disassembling
from capstone import *
def disas(code, address):
    md = Cs(CS_ARCH_X86, CS_MODE_32)
    insn = md.disasm(str(code), address)
    for i in insn:
        print ('0x%x: %s\t%s' %(i.address, i.mnemonic, i.op_str))


# hook WinAPI symbols
def hook_code(uc, address, size, user_data):
    global DEBUG
    #print("hooking %x" %address)
    if DEBUG:
        #code disassembly
        # read this instruction code from memory
        code = uc.mem_read(address, size)
        disas(code, address)

    esp = uc.reg_read(UC_X86_REG_ESP)

    if address in utils.import_symbols:
        # print(">> CALL HOOK API at %x" %address)
        globals()['hook_' + utils.import_symbols[address]](uc, address, esp)


def hook_mem_error(uc, type, addr,*args):
    print(">> ERROR: unmapped memory access at 0x%x" %addr)
    return False


def sandbox():
    global DEBUG
    from optparse import OptionParser
    usage = "Usage: %prog [options] filename"
    parser = OptionParser(usage)
    parser.add_option("-d", "--debug",
                      action="store_true", dest="debug")
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.print_help()
        return

    DEBUG = options.debug

    print('>> Emulating Win32 shellcode ...')
    try:
        uc = Uc(UC_ARCH_X86, UC_MODE_32)
        uc.hook_add(UC_HOOK_MEM_UNMAPPED, hook_mem_error)

        # setup stack memory
        uc.mem_map(STACK_ADDR, STACK_SIZE)
        uc.reg_write(UC_X86_REG_ESP, STACK_ADDR + 0x3000)
        uc.reg_write(UC_X86_REG_EBP, STACK_ADDR + 0x3000)

        # load shellcode in
        uc.mem_map(CODE_ADDR, CODE_SIZE)
        shellcode = read_shellcode(args[0])
        uc.mem_write(CODE_ADDR, shellcode)

        # setup GDT & FS
        setup_gdt_segment(uc, GDT_ADDR, GDT_LIMIT, UC_X86_REG_FS, 1, FS_ADDR, FS_SIZE, init = True)

        # setup Windows environment
        setup_win32_xp(uc, FS_ADDR)
```

Keystone Engine

NGUYEN Anh Quynh <aquynh -at- gmail.com>

http://www.keystone-engine.org

What's Wrong with Assembler



- Nothing is up to our standard, even in 2016!
  - Yasm: X86 only, no longer updated
  - Intel XED: X86 only, miss many instructions & closed-source
  - Use assembler to generate object files
  - Other important archs: Arm, Arm64, Mips, PPC, Sparc, etc?

What do we need?

- Multiple archs: x86, ARM+ ARM64 + Mips + PPC and more
- Multiple platform: Windows, Linux, OSX and more
- Multiple binding: Python, Ruby, Java, C# and more

- Clean, simple, intuitive & architecture-neutral API
- Provide break-down details on instructions
- Friendly license: BSD

Lots of Work !

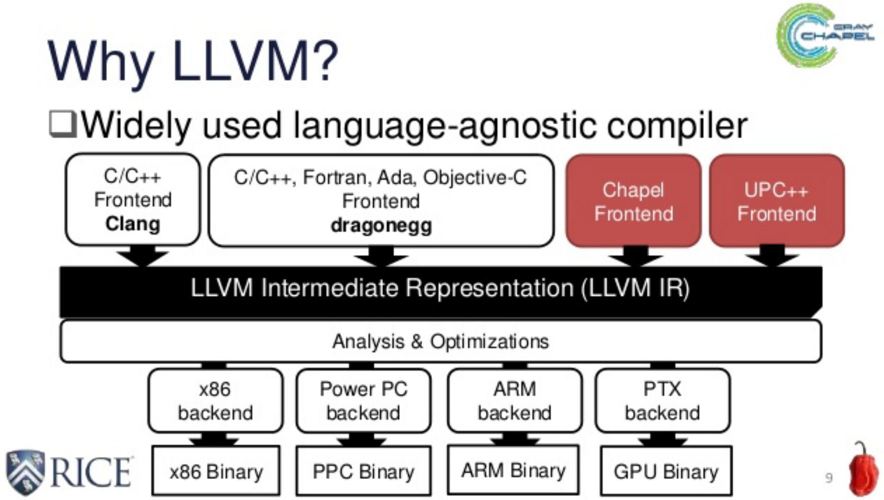
- Multiple archs: x86, ARM
- Actively maintained & update within latest arch's change
- Multiple platform: Windows, Linux
- Understanding opcode, Intel x86 it self with 1500++ documented instructions

- Support python and ruby as binding languages
- Single man show
- Target finish within 12 months

A Good Assembler

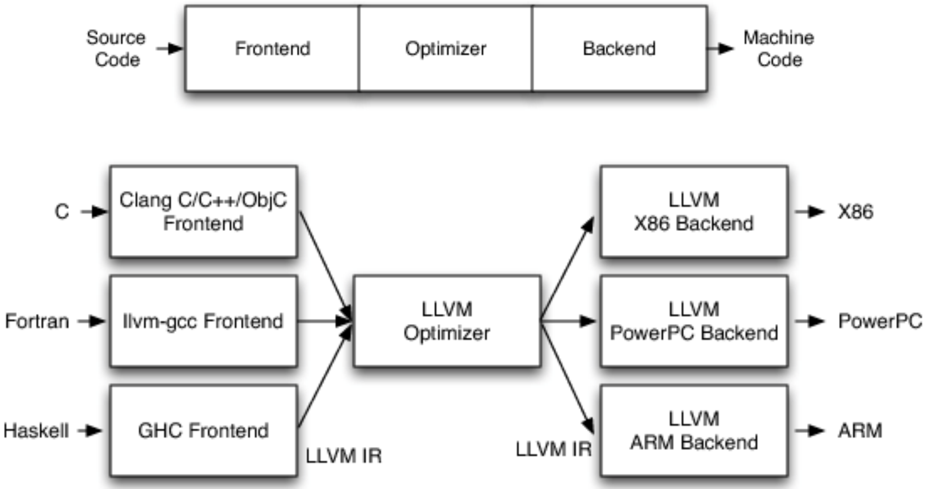
- Multiple archs: x86, ARM
- Actively maintained & update within latest arch's change
- Multiple platform: Windows, Linux
- Support python and ruby as binding languages
- Friendly license (BSD)
- Easy to setup

Not Reinventing the Wheel



- Open source project compiler
- Sets of modules for machine code representing, compiling, optimizing
- Backed by many major players: AMD, Apple, Google, Intel, IBM, ARM, Imgtec, Nvidia, Qualcomm, Samsung, etc
- Incredibly huge (compiler) community around.

Fork from LLVM



- Multiple architectures ready
- In-build assembler (MC module)
  - Only, Only and Only build for LLVM
  - actively maintained
- Very actively maintained & updated by a huge community

Are We Done

Issue 1

- LLVM not just assembler, but also disassembler, bitcode, InstPrinter, Linker Optimization, etc
- LLVM codebase is huge and mixed like spaghetti

Solutions

- Keep only assembler code & remove everything else unrelated
- Rewrites some components but keep AsmParser, CodeEmitter & AsmBackend code intact (so easy to sync with LLVM in future, e.g. update)
- Keep all the code in C++ to ease the job (unlike Capstone)
  - No need to rewrite complicated parsers
  - No need to fork llvm-tblgen

Issue 2

- LLVM compiled into multiple libraries
  - Supported libs
  - Parser
  - TableGen and etc
- Keystone needs to be a single library

Solutions

- Modify linking setup to generate a single library
  - libkeystone.[so, dylib] + libkeystone.a
  - keystone.dll + keystone.lib

Are We Done

Issue 3

- Relocation object code generated for linking in the final code generation phase of compiler
- Ex on X86:
  - inc [_var1] → 0xff, 0x04, 0x25, A, A, A, A

Solutions

- Make fixup phase to detect & report missing symbols
- Propagate this error back to the top level API ks_asm()

Issue 4

- Ex on ARM: blx 0x86535200 → 0x35, 0xf1, 0x00, 0xe1

Solutions

- ks_asm() allows to specify address of first instruction
- Change the core to retain address for each statement
- Find all relative branch instruction to fix the encoding according to current & target address

Are We Done

Issue 5

- Ex on X86: vaddpd zmm1, zmm1, zmm1, x → "this is not an immediate"
- Returned llvm_unreachable() on input it cannot handle

Solutions

- Fix all exits & propagate errors back to ks_asm()
    - Parse phase
    - Code emit phase

Issue 6

- LLVM does not support non-LLVM syntax
    - We want other syntaxes like Nasm, Masm, etc
- Bindings must be built from scratch
- Keep up with upstream code once forking LLVM to maintain ourselves

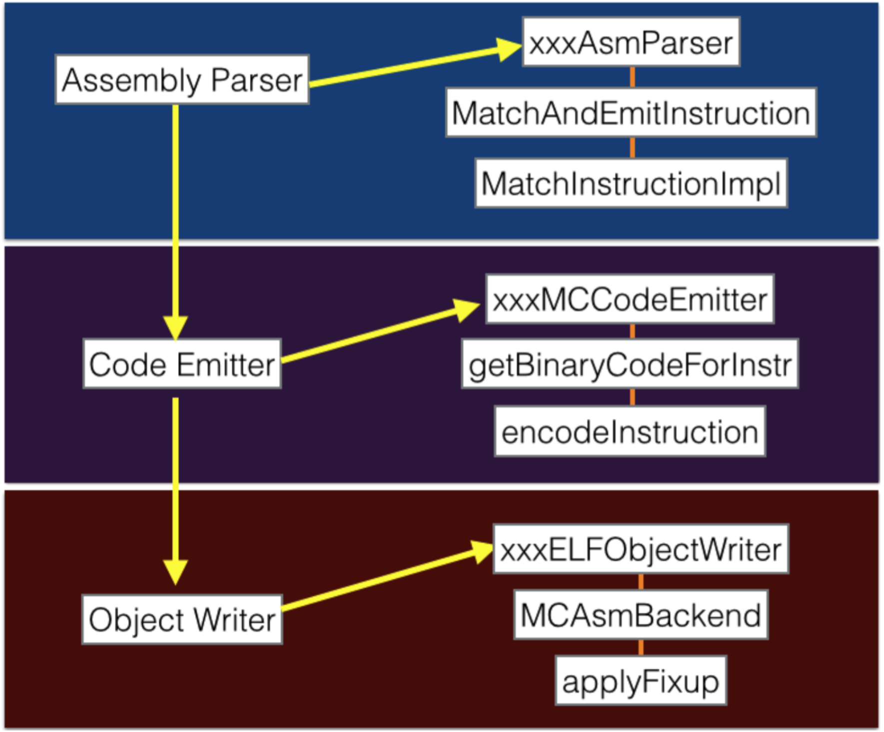
Solutions

- Extend X86 parser for new syntaxes: Nasm, Masm, etc
- Built Python binding
- Extra bindings came later, by community: NodeJS, Ruby, Go, Rust, Haskell & OCaml
- Keep syncing with LLVM upstream for important changes & bug-fixes

Keystone is not LLVM

Fork and Beyond

- Independent & truly a framework
  - Do not give up on bad-formed assembly
- Aware of current code position (for relative branches)
- Much more compact in size, lightweight in memory
- Thread-safe with multiple architectures supported in a single binary More flexible: support X86 Nasm syntax
- Support undocumented instructions: X86
- Provide bindings (Python, NodeJS, Ruby, Go, Rust, Haskell, OCaml as of August 2016)



Demo

```c
/* test1.c */
#include <stdio.h>
#include <keystone/keystone.h>

// separate assembly instructions by ; or \n
#define CODE "INC ecx; DEC edx"

int main(int argc, char **argv)
{
    ks_engine *ks;
    ks_err err;
    size_t count;
    unsigned char *encode;
    size_t size;

    err = ks_open(KS_ARCH_X86, KS_MODE_32, &ks);
    if (err != KS_ERR_OK) {
        printf("ERROR: failed on ks_open(), quit\n");
        return -1;
    }

    if (ks_asm(ks, CODE, 0, &encode, &size, &count) != KS_ERR_OK) {
        printf("ERROR: ks_asm() failed & count = %lu, error = %u\n",
               count, ks_errno(ks));
    } else {
        size_t i;

        printf("%s = ", CODE);
        for (i = 0; i < size; i++) {
            printf("%02x ", encode[i]);
        }
        printf("\n");
        printf("Compiled: %lu bytes, statements: %lu\n", size, count);
    }

    // NOTE: free encode after usage to avoid leaking memory
    ks_free(encode);

    // close Keystone instance when done
    ks_close(ks);

    return 0;
}
```

```
$ make
cc -o test1 test1.c -lkeystone -lstdc++ -lm

$ ./test1
INC ecx; DEC edx = 41 4a
Compiled: 2 bytes, statements: 2
```

```python
from keystone import *

# separate assembly instructions by ; or \n
CODE = b"INC ecx; DEC edx"

try:
  # Initialize engine in X86-32bit mode
  ks = Ks(KS_ARCH_X86, KS_MODE_32)
  encoding, count = ks.asm(CODE)
  print("%s = %s (number of statements: %u)" %(CODE, encoding, count))
except KsError as e:
  print("ERROR: %s" %e)
```

```
$ ./test1.py
INC ecx; DEC edx = [65, 74] (number of statements: 2)
```

Show Case: metame

Before

```
,=< 0x10012e09      eb10           jmp 0x10012e1b                ;[
|   ; JMP XREF from 0x10012d82 (fcn.100124ed)
|   0x10012e0b      8b542410       mov edx, dword [esp + 0x10] ;
|   0x10012e0f      8d4bff         lea ecx, [ebx - 1]
|   0x10012e12      51             push ecx
|   0x10012e13      52             push edx
|   0x10012e14      8bce           mov ecx, esi
|   0x10012e16      e807eeffff     call fcn.10011c22             ;[
|   ; JMP XREF from 0x10012622 (fcn.100124ed)
|   ; JMP XREF from 0x10012d79 (fcn.100124ed)
|   ; JMP XREF from 0x10012de3 (fcn.100124ed)
|   ; JMP XREF from 0x10012e09 (fcn.100124ed)
`-> 0x10012e1b      8b7c2454       mov edi, dword [esp + 0x54] ;
    0x10012e1f      bd01000000     mov ebp, 1
    0x10012e24      3bdf           cmp ebx, edi
,=< 0x10012e26      7321           jae 0x10012e49                ;[
|   0x10012e28      8d9b00000000   lea ebx, [ebx]
|   ; JMP XREF from 0x10012e47 (fcn.100124ed)
```

```
,=< 0x080cbd91      eb0d           jmp 0x80cbda0
|   0x080cbd93      90             nop
|   0x080cbd94      90             nop
|   0x080cbd95      90             nop
|   0x080cbd96      90             nop
|   0x080cbd97      90             nop
|   0x080cbd98      90             nop
|   0x080cbd99      90             nop
|   0x080cbd9a      90             nop
|   0x080cbd9b      90             nop
|   0x080cbd9c      90             nop
|   0x080cbd9d      90             nop
|   0x080cbd9e      90             nop
|   0x080cbd9f      90             nop
`-> 0x080cbda0      55             push ebp
    0x080cbda1      89e5           mov ebp, esp
    0x080cbda3      57             push edi
    0x080cbda4      89c7           mov edi, eax
    0x080cbda6      56             push esi
```

After

```
,=< 0x10012e09      eb10           jmp 0x10012e1b
|   ; JMP XREF from 0x10012d82 (fcn.100124ed)
|   0x10012e0b      8b542410       mov edx, dword [esp + 0x10]
|   0x10012e0f      8d4bff         lea ecx, [ebx - 1]
|   0x10012e12      51             push ecx
|   0x10012e13      52             push edx
|   0x10012e14      56             push esi
|   0x10012e15      59             pop ecx
|   0x10012e16      e807eeffff     call fcn.10011c22
|   ; JMP XREF from 0x10012622 (fcn.100124ed)
|   ; JMP XREF from 0x10012d79 (fcn.100124ed)
|   ; JMP XREF from 0x10012de3 (fcn.100124ed)
|   ; JMP XREF from 0x10012e09 (fcn.100124ed)
`-> 0x10012e1b      8b7c2454       mov edi, dword [esp + 0x54]
    0x10012e1f      9c             pushfd
    0x10012e20      31ed           xor ebp, ebp
    0x10012e22      45             inc ebp
    0x10012e23      9d             popfd
```

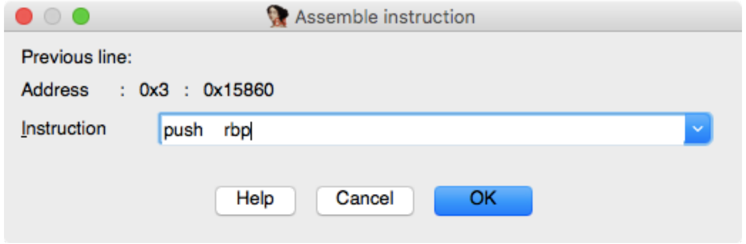
```
   ,=< 0x080cbd91      eb0d           jmp 0x80cbda0
  ,==< 0x080cbd93      eb01           jmp 0x80cbd96
  ||   0x080cbd95      42             inc edx
 ,`--> 0x080cbd96      eb01           jmp 0x80cbd99
 | |   0x080cbd98      5a             pop edx
 `,==< 0x080cbd99      eb01           jmp 0x80cbd9c
  ||   0x080cbd9b      5f             pop edi
 ,`--> 0x080cbd9c      eb01           jmp 0x80cbd9f
 | |   0x080cbd9e      40             inc eax
 `---> 0x080cbd9f      90             nop
   `-> 0x080cbda0      55             push ebp
       0x080cbda1      54             push esp
       0x080cbda2      5d             pop ebp
       0x080cbda3      57             push edi
       0x080cbda4      50             push eax
       0x080cbda5      5f             pop edi
       0x080cbda6      56             push esi
       0x080cbda7      53             push ebx
       0x080cbda8      83ec2c         sub esp, 0x2c
```

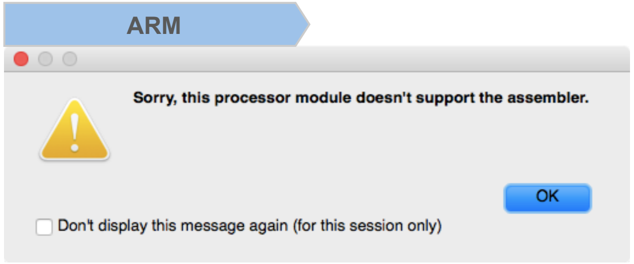
One More Thing

The IDA Pro

IDA Pro

- RE Standard
- Patching on the fly is always a must
- Broken “Edit\Patch Program\ Assembler” is always giving us problem

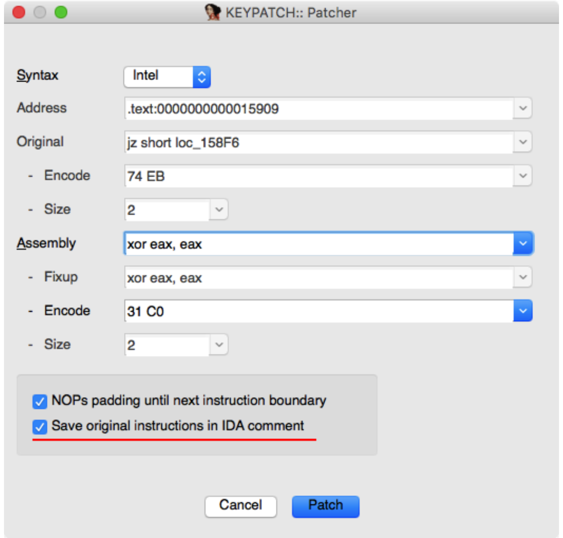


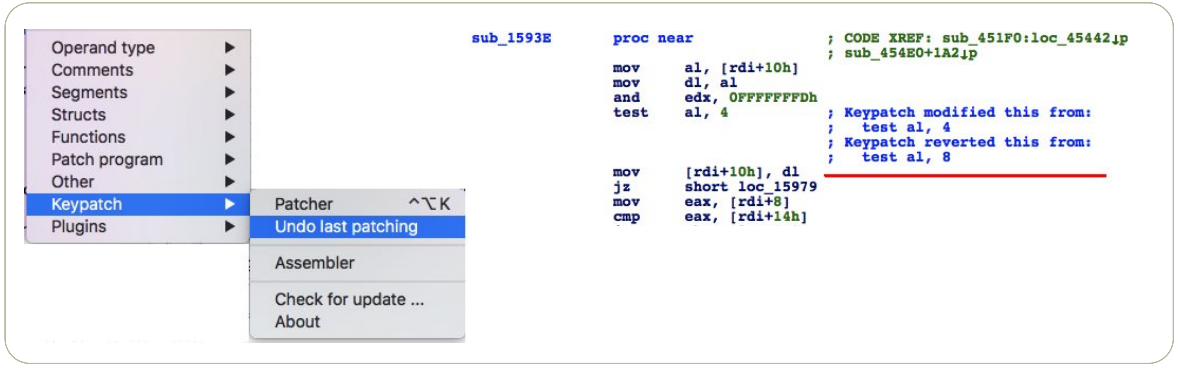


Keypatch

A binary editor plugin for IDA Pro

- Fully open source @ https://keystone-engine.org/keypatch
- On the fly patching in IDA Pro with Multi Arch
- Base on Keystone Engine
- By Nguyen Anh Quynh & Thanh Nguyen (rd) from vnsecurity.net

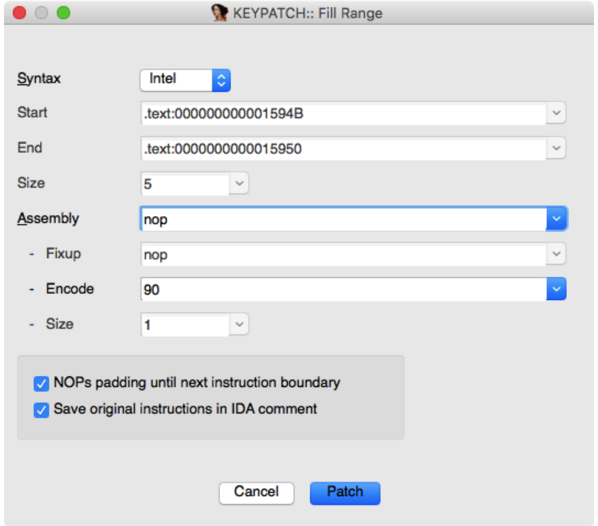


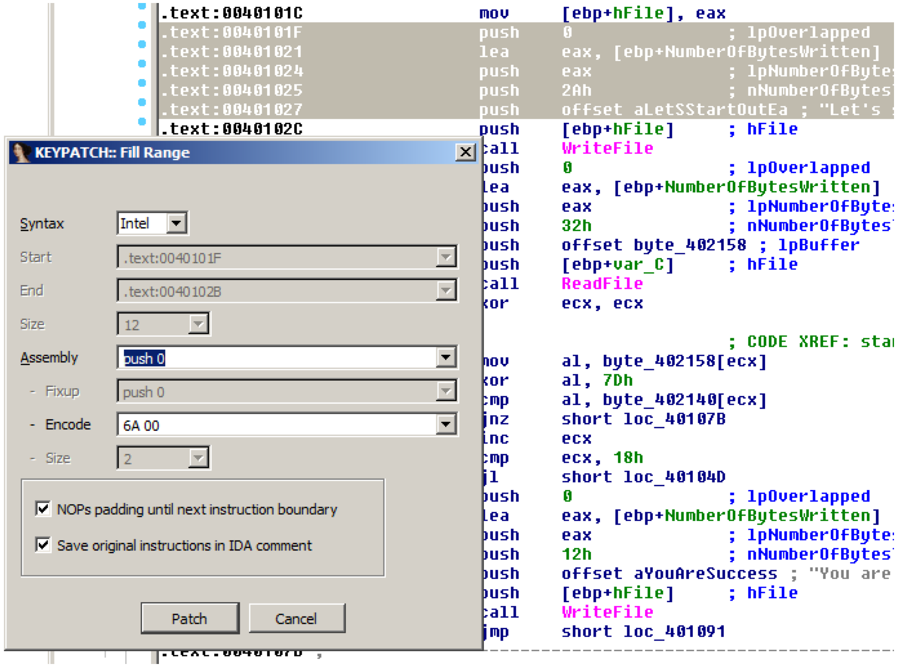


Latest Keypatch and DEMO

Fill Range

- Select Start, End range and patch with bytes
- Goto: Edit | Keypatch | Fill Range
- QQ: 2880139049





THANKS

[Hacker@KCon]