fireeye.com

Writing a libemu/Unicorn Compatability Layer

by David Zimmer

5-6 minutes

In this post we are going to take a quick look at what it takes to write a libemu compatibility layer for the Unicorn engine. In the course of this work, we will also import the libemu Win32 environment to run under Unicorn.

For a bit of background, libemu is a lightweight x86 emulator written in C by Paul Baecher and Markus Koetter. It was released in 2007 and includes a built-in Win32 environment that allows shellcodes to resolve API at runtime. The library also provides end users with a convenient way to receive callbacks when API functions are hit. The original project supported 5 Windows dlls, 51 hooks and 234 opcodes all wrapped in a tight 1mb package. Unfortunately it is no longer being updated.

In late 2015, we saw the Unicorn engine project released by Nguyen Anh Quynh and Dang Hoang Vu. This project takes the processor emulators from QEMU and wraps them into an easy to use library. Unicorn, however, does not provide a Win32 layer.

As an experiment, we were curious to see what it would take to bring the libemu Win32 environment into Unicorn. This task

actually turned out to be quite simple since it was nicely self contained. In the process of exploring this it also made sense to write a basic shim layer to support the libemu API and translate its inner workings over to Unicorn.

Lets start with the common libemu API:

```
//emu_memory.h
/* read access, these functions return -1 on error  */
int32_t emu_memory_read_byte(struct emu_memory *m, uint32_t addr, uint8_t *byte);
int32_t emu_memory_read_block(struct emu_memory *m, uint32_t addr, void *dest,
size_t len);
int32_t emu_memory_read_word(struct emu_memory *m, uint32_t addr, uint16_t *word);
int32_t emu_memory_read_dword(struct emu_memory *m, uint32_t addr, uint32_t *dword);
int32_t emu_memory_read_string(struct emu_memory *m, uint32_t addr, struct
emu_string *s, uint32_t maxsize);
int32_t emu_memory_read_wide_string(struct emu_memory *m, uint32_t addr, struct
emu_string *s, uint32_t maxsize);

int32_t emu_memory_write_byte(struct emu_memory *m, uint32_t addr, uint8_t byte);
int32_t emu_memory_write_block(struct emu_memory *m, uint32_t addr, void *src,
size_t len);
int32_t emu_memory_write_word(struct emu_memory *m, uint32_t addr, uint16_t word);
int32_t emu_memory_write_dword(struct emu_memory *m, uint32_t addr, uint32_t dword);

//emu.h
struct emu_cpu *emu_cpu_get(struct emu *e);
struct emu_memory *emu_memory_get(struct emu *e);


//emu_cpu.h
enum emu_reg32 {
        eax = 0, ecx, edx, ebx, esp, ebp, esi, edi
};

uint32_t emu_cpu_reg32_get(struct emu_cpu *cpu_p, enum emu_reg32 reg);
void emu_cpu_reg32_set(struct emu_cpu *cpu_p, enum emu_reg32 reg, uint32_t val);

uint16_t emu_cpu_reg16_get(struct emu_cpu *cpu_p, enum emu_reg16 reg);
void emu_cpu_reg16_set(struct emu_cpu *cpu_p, enum emu_reg16 reg, uint16_t val);

uint8_t emu_cpu_reg8_get(struct emu_cpu *cpu_p, enum emu_reg8 reg);
void emu_cpu_reg8_set(struct emu_cpu *cpu_p, enum emu_reg8 reg, uint8_t val);

uint32_t emu_cpu_eflags_get(struct emu_cpu *c);
void emu_cpu_eflags_set(struct emu_cpu *c, uint32_t val);

void emu_cpu_eip_set(struct emu_cpu *c, uint32_t eip);
uint32_t emu_cpu_eip_get(struct emu_cpu *c);
```

The API is actually very similar to Unicorn:

```
uc_err uc_reg_write(uc_engine *uc, int regid, const void *value);
uc_err uc_reg_read(uc_engine *uc, int regid, void *value);
uc_err uc_mem_write(uc_engine *uc, uint64_t address, const void *bytes, size_t
```

```
uc_err uc_mem_write(uc_engine *uc, uint64_t address, const void *bytes, size_t
size);
uc_err uc_mem_read(uc_engine *uc, uint64_t address, void *bytes, size_t size);
uc_err uc_mem_map(uc_engine *uc, uint64_t address, size_t size, uint32_t perms);
uc_err uc_mem_map_ptr(uc_engine *uc, uint64_t address, size_t size, uint32_t perms,
void *ptr);
uc_err uc_mem_unmap(uc_engine *uc, uint64_t address, size_t size);
uc_err uc_mem_protect(uc_engine *uc, uint64_t address, size_t size, uint32_t perms);
```

The major differences are that Unicorn does everything through an opaque uc_engine* handle, while libemu uses a series of structs such as emu, emu_cpu, and emu_memory:

```
//emu_cpu.c
struct emu
{
        ...
        struct emu_memory *memory;
        struct emu_cpu *cpu;
        ...
};

//emu_cpu_data.h
struct emu_cpu
{
        struct emu *emu;
        struct emu_memory *mem;
        ...
        uint32_t eip;
        uint32_t eflags;
        uint32_t reg[8];
        ...
}
```

In general, the emu and emu_memory structures are passed directly as arguments to API wrappers such as emu_cpu_get, emu_memory_get and the emu_memory_read/write functions. There is one common case of direct member access to the emu_cpu structure that requires some special attention. This structure gives the user direct read/write access to the emulator’s virtual processor and is commonly utilized by user code. Examples to support include:

```
emu_cpu_get(e)->eip
cpu->eflags = x
x = cpu->reg[eax]
cpu->reg[esp] -= 4;
```

The next task was to see if we could mimic the direct access to the emu_cpu elements as if they were static struct fields. Here we enter the world of C++ operator overloading.

```
//this class traps int value gets/sets so we can do dynamic things as they are
accessed...
class CAccessCheck
{
    int   index;
    int   role;
    uc_engine* uc;

    public:
        CAccessCheck(void): index(0), role(0), uc(0){}
        CAccessCheck(int r,uc_engine* engine):index(0), role(r), uc(engine){}
        CAccessCheck(int i, int r,uc_engine* engine): index(i), role(r), uc(engine)
{}

    //we are setting the value..
    void operator=(uint32_t v);

    //we are accessing the value. note if in a printf you MUST cast to (int)
    operator uint32_t const();

    //support the += and -= operations
    uint32_t operator +=(uint32_t v){
        uint32_t tmp;
        tmp = operator uint32_t const();
        tmp += v;
        operator=(tmp);
        return tmp;
    }

    uint32_t operator -=(uint32_t v){
        uint32_t tmp;
        tmp = operator uint32_t const();
        tmp -= v;
        operator=(tmp);
        return tmp;
    }

};

//this class activates on use of the [] operators to mimic direct array access
class CRegAccess{
  protected:
    int m_mode;
    uc_engine* uc;

   public:
     CRegAccess(void){m_mode=0;uc=0;};
     CRegAccess(int mode,uc_engine* engine){m_mode = mode; uc=engine;}
     CAccessCheck operator[](int index){
          return CAccessCheck(index, this->m_mode, this->uc);
     }

 };
```

```
class emu_cpu {
    public:
        uc_engine* uc;
        uc_engine* mem;
        CAccessCheck eip;
        CAccessCheck eflags;
        CRegAccess reg;
        //CRegAccess reg16;
        //CRegAccess reg8;
        emu_cpu(uc_engine* engine);
};

void CAccessCheck::operator=(uint32_t v)
{
    if(role==1){ //eip
        emu_cpu_eip_set(this->uc,v);
    }
    else if(role==2){ //eflags
        uc_reg_write(this->uc,UC_X86_REG_EFLAGS,&v);
    }
    else if(role==32){ //32bit register access
        emu_reg32_write(this->uc,(emu_reg32)index,v);
    }

    //printf("SET index: %d   value: %d   role:%d\n",index,v,role);
}

CAccessCheck::operator uint32_t const()
{
    int ret;

    if(role==1){ //eip
        ret = emu_cpu_eip_get(this->uc);
    }
    else if(role==2){ //eflags
        uc_reg_read(this->uc,UC_X86_REG_EFLAGS,&ret);


    }
    else if(role==32){ //32bit register access
        ret = emu_reg32_read(this->uc,(emu_reg32)index);
    }

    //printf("GET  index: %d   role:=%d\n", index, role);
    return ret;
}

emu_cpu::emu_cpu(uc_engine* engine){
    this->uc = engine;
    this->mem = engine;
    eip = CAccessCheck(1,engine);
    eflags = CAccessCheck(2,engine);
    reg = CRegAccess(32,engine);
    //reg16 = CRegAccess(16,engine);
    //reg8 = CRegAccess(8,engine);
}

emu_cpu *emu_cpu_get(uc_engine *uc){
    return new emu_cpu(uc);
}
```

```
uc_engine *emu_memory_get(uc_engine *uc){
    return uc;
}
```

With these tasks complete, porting existing code from libemu over to Unicorn should be a pretty straightforward task.

In Figure 1 we see an initial test, we put together that includes the Win32 environment, shim layer, several API hooks and a hard coded payload.

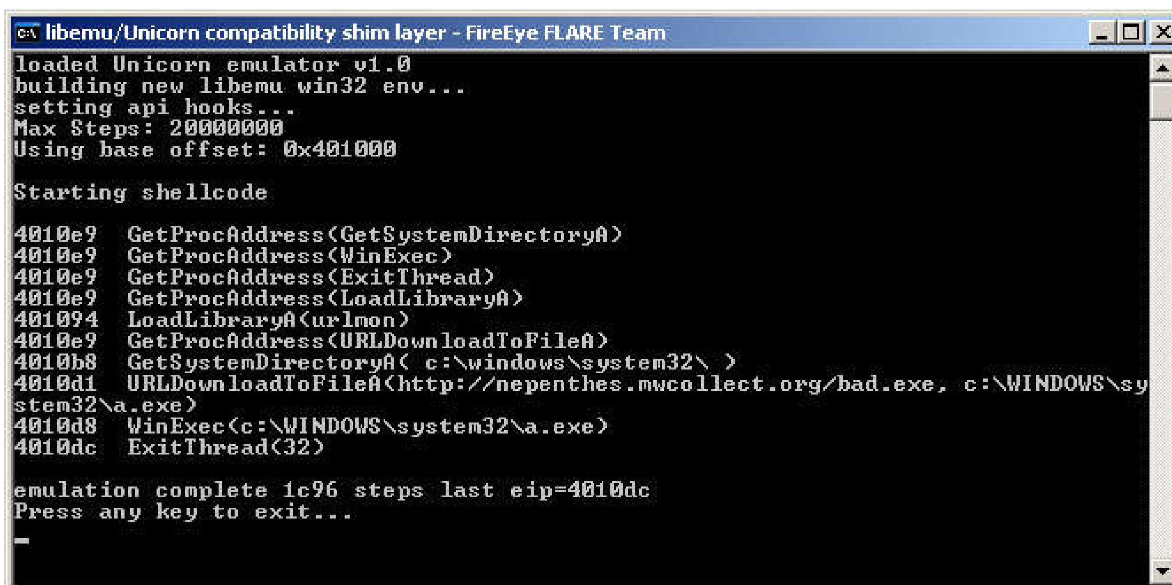

Figure 1: Initial test of the libemu Win32 environment and hooks running under Unicorn

With this working, the next stage was to try it out against a larger code base. Here we imported the userhooks.cpp from scdbg, an extension of the libemu sctest that includes some 250 API hooks. As it turns out, very few changes were required to get it working.

In Figure 2, we can see the results of testing it against a fairly complex shellcode that:

- allocates virtual memory
- copies code to the new alloc

- creates a new thread
- downloads an executable
- checks the registry for the presence of Antivirus software

Note that while this shellcode would normally do process injection, scdbg handles it all inline for simplified analysis.

```
cmd

D:\unicorn_libemu\bin> scdbg -f VirtualAllocEx.sc -u

loaded Unicorn emulator v1.0
building new libemu win32 env...
Loaded b30 bytes from file VirtualAllocEx.sc
Max Steps: -1
Using base offset: 0x401000

Starting shellcode

401431   LoadLibraryA(kernel32)
401431   LoadLibraryA(user32)
401431   LoadLibraryA(advapi32)
401431   LoadLibraryA(ntdll)
40115a   FindWindowA(class=Progman, window=Program Manager)
40116a   GetWindowThreadProcessId(h=0, buf=12ffd0)
40117a   OpenProcess(access=1f0fff, inherit=0, pid=14077ac0) - Process:
401194   VirtualAllocEx(pid=14077ac0, base=0 , sz=1000) = 600000
4011b1   WriteProcessMemory(pid=14077ac0, base=600000 , buf=4017c3, sz=310, written=0)
4011c7   CreateRemoteThread(pid=14077ac0, addr=600000 , arg=0, flags=0, *id=0)
         Transferring execution to threadstart...
600190   LoadLibraryA(kernel32)
600190   LoadLibraryA(ntdll)
600190   LoadLibraryA(urlmon)
60004f   Sleep(0x15f90)
         Allocation 104 < 1024 adjusting...
60005c   GlobalAlloc(sz=400) = 601000
60006b   GetTempPathA(len=104, buf=601000) = 8
60007b   strcat(d:\temp\, vss0000001.exe)
600084   DeleteFileA(d:\temp\vss0000001.exe)
60009a   URLDownloadToFileA(http://ojymo.com/css/ac/s.exe, d:\temp\vss0000001.exe)
6000b5   CreateFileA(d:\temp\vss0000001.exe) = 4
6000cd   GetFileSize(4, 0) = ffffffff
         Allocation 104 < 1024 adjusting...
4011e1   GlobalAlloc(sz=400) = 602000
40122b   RegOpenKeyExA(HKLM\, SOFTWARE\\AhnLab\\V3Lite)
         Allocation 104 < 1024 adjusting...
4011e1   GlobalAlloc(sz=400) = 603000
40122b   RegOpenKeyExA(HKLM\, SOFTWARE\\AhnLab\\V3 365 Clinic)
         Allocation 104 < 1024 adjusting...
4011e1   GlobalAlloc(sz=400) = 604000
40122b   RegOpenKeyExA(HKLM\, SOFTWARE\\NHN Corporation\\NaverVaccine)
         Allocation 104 < 1024 adjusting...
4011e1   GlobalAlloc(sz=400) = 605000
40122b   RegOpenKeyExA(HKLM\, SOFTWARE\\ESTsoft\\ALYac)
40108b   ExitProcess(0)

emulation complete ed9b66 steps last eip=40108b
```

Figure 2: Complex shellcode running with hooks imported from scdbg

Another large feature to test was the scdbg debug shell. When testing software in an emulated environment, having interactive debug tools available is extremely handy.

Figure 3 shows an example of setting a breakpoint, single stepping,

and examining memory of code running in the emulator.

```
scdbg - http://sandsprite.com

D:\unicorn_libemu\bin>scdbg -f fire.sc -bp 4010c1

loaded Unicorn emulator v1.0
building new libemu win32 env...
Breakpoint 0 set at 4010c1
Loaded 1c7 bytes from file fire.sc
Max Steps: 2000000
Using base offset: 0x401000

Starting shellcode

4010a3  GetTempPathA(len=ff, buf=4011c7) = 8
        Breakpoint 0 hit at: 0
4010c1   682C5B06E2              push dword 0xe2065b2c        step: 224498
eax=4011c7    ecx=4010a3    edx=e449f330  ebx=4011c7
esp=12fff8    ebp=401006    esi=0         edi=0         eip=4010c1

dbg>
4010c6   FFD5                    call ebp              step: 224499  foffset:
eax=4011c7    ecx=4010a3    edx=e449f330  ebx=4011c7
esp=12fff4    ebp=401006    esi=0         edi=0         eip=4010c6

dbg> Enter hex base to dump: (hex/reg) 0xeax
4011c7
Enter hex size: (hex/reg) 0x20
20

         0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
4011c7   64 3a 5c 74 65 6d 70 5c  2e 63 6f 6d 00 00 00 00      d:\temp\.com....
4011d7   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00      ................

dbg>
```

Figure 3: Imported scdbg debug shell running with Unicorn Engine and libemu shim layer

Conclusion

In this article we took a quick look at the differences between the libemu and Unicorn emulators API. This allowed us to create a shim layer to import legacy libemu code and use it with Unicorn largely unchanged.

Once the shim layer was in place, we next imported the libemu Win32 Environment so we could run it under Unicorn.

As a final test we ported several large portions of the scdbg project, which was originally written to run under libemu. Here our previous work allowed for the importation of scdbg's 250+ API hooks and debug shell to run under Unicorn with only minimal changes.

Overall the entire process went quite smoothly and should provide

benefits for developers of libemu and/or Unicorn. If you would like to experiment for yourself you can download a copy of our test project here.