# Micro Execution

ICSE 2014

Patrice Godefroid

김영철

2019. 5. 23.

# Micro Execution

- 코드 조각을 실행시키는 기술
  - 유저가 작성하는 테스트 드라이버 코드 / 입력 데이터 없이 작동

- 가상 머신(MicroX)의 존재
  - 코드 조각을 충돌 없이 실행시키는 도구
  - 입력 값이 필요할 때 자동으로 생성

# Introduction

- unit testing
  - its role is to detect errors in the component's logic, check all corner cases, and provide 100% code coverage.
  - is so hard and expensive to perform, especially for components of large, complex, legacy code bases.
- propose Micro Execution

# Introduction

- Micro Execution's scenario
  - The user simply identifies a function or code location in exe an or dll.
  - A runtime VM starts executing the code at that location
    catches all memory operations before they occur
    allocates memory in order to perform memory op
    provides input values according to a memory policy

* memory policy defines what read memory accesses should be treated as input

# Introduction

- MicroX
  - is a VM allowing micro execution of x86 binary code.
  - is a modification of the Nirvana VM, iDNA, TruScan and SAGE.
    - SAGE is the whitebox fuzzer using dynamic symbolic execution, constraint generation and solving techniques.
  - not require for writing any test-driver code.

# Overview

- What is Micro Execution?
  (1) select code location & test generation mode
  (2) execute the first instruction of function foo
  (3) VM detects that the execution wants to read a 4-byte value for p
  ➔ p is an input according to the below memory policy
  (4) a 4-byte value is generated for p and is returned to the program
  …
  (n) After the execution of the return statement, it terminates.

An input is defined as any value read from an uninitialized function argument, or from a dereference of a previous input used as an address (recursive definition). (*)

<memory policy>

```
void foo(char *p) { // p is a 4-byte input
  char v = *p;      // *p is a 1-byte input
  return;
}
```

<selected location>

# Overview

- How is Micro Execution performed with MicroX?
  - MicroX executes x86 Inst one by one.
  - It catches all memory operations.

    the expression [ebp] is to access memory

    line 1,2 are not memory access

```
    [...]
1:  push  ebp              ; foo starts here
2:  mov   ebp, esp
3:  push  ecx
4:  mov   eax, DWORD PTR [ebp+8]  ; p
5:  mov   cl, BYTE PTR [eax]      ; *p
6:  mov   BYTE PTR [ebp-1], cl    ; v
7:  mov   esp, ebp
8:  pop   ebp
9:  ret   0
    [...]
```

  - at line 4, detect memory access

    [ebp+8] is function argument, thus an input

    allocates a 4-byte buffer at address X in an external memory

    maps the ebp+8 with X

  - In next section, we discuss in detail.

# Overview

- How is Micro Execution performed with MicroX?
  - MicroX executes x86 Inst one by one.
  - It catches all memory operations.

    the expression [ebp] is to access memory

    line 1,2 are not memory access

  - at line 4, detect memory access

    [ebp+8] is function argument, thus an input

    allocates a 4-byte buffer at address X in an external memory

    maps the ebp+8 with X

  - In next section, we discuss in detail.

```
    [...]
1:  push  ebp                 ; foo starts here
2:  mov   ebp, esp
3:  push  ecx
4:  mov   eax, DWORD PTR [ [X] ] ; p
5:  mov   cl, BYTE PTR [eax]     ; *p
6:  mov   BYTE PTR [ebp-1], cl   ; v
7:  mov   esp, ebp
8:  pop   ebp
9:  ret   0
    [...]
```

# Overview

- Report of Micro Execution

```
[...]
push  ebp               ; foo starts here
mov   ebp, esp
push  ecx
mov   eax, DWORD PTR [ebp+8] ; p
```

  - line 3, memory access [ebp+8]
  - line 5, 4-byte value random generation
    - 이 시점에서, 포인터로 사용될지 아직 모름.
      - add to list of known addresses (line 6)
    - is stored in a 4-byte buffer located at X (line 7)
    - ebp+8 (visible) is mapped to the X (invisible)
    - eax = [ebp+8] = [X] = 0x00201478

```
1:  initEIP is 72B51005
2:  initEBP is 001EF988

3:  Read Mem Access at address 001EF990 of 4 bytes
4:      Initializing 4 input bytes:
5:        [0]=78 [1]=14 [2]=20 [3]=00
6:      Adding 00201478 to list of known addresses
7:    SetGuestEffectiveAddress returned 00201440

8:  Read Mem Access at address 00201478 of 1 bytes
9:      Initializing 1 input bytes: [0]=29
10:   SetGuestEffectiveAddress returned 0020C490

11: Write Mem Access at address 001EF987 of 1 bytes
12:   SetGuestEffectiveAddress returned 001EF987

13: END: ExitProcess is called

14: ***** External Memory Stats: *****
15: Number of Mem Accesses: 2 (2 Reads, 0 Writes)
16: Number of Addresses: 2 (total 5 bytes)
17: Number of Inputs: 2 (total 5 bytes)

18: ***** Native Memory Stats: *****
19: Number of Module Accesses: 0 (0 Reads, 0 Writes)
20: Number of Other Accesses: 1 (0 Reads, 1 Writes)

21: ***** General Stats: *****
22: Number of Unique Instructions After Start: 9
23: Number of Warnings: 0
24: Number of Errors: 0
```

# Overview

- Report of Micro Execution

```
push   ebp              ; foo starts here
mov    ebp, esp
push   ecx
mov    eax, DWORD PTR [ebp+8]  ; p
mov    cl, BYTE PTR [eax]      ; *p
```

- line 8, memory access [eax]
  - is an input by memory policy
- line 9, 1-byte value random generation
  - is stored in located at 0x20C490 (line 10)
  - [eax] is mapped to the 0x0020C490 = 0x29
  - cl = byte ptr [eax] = address 0x0020C490 = 0x29

```
1:  initEIP is 72B51005
2:  initEBP is 001EF988

3:  Read Mem Access at address 001EF990 of 4 bytes
4:      Initializing 4 input bytes:
5:        [0]=78 [1]=14 [2]=20 [3]=00
6:      Adding 00201478 to list of known addresses
7:    SetGuestEffectiveAddress returned 00201440

8:  Read Mem Access at address 00201478 of 1 bytes
9:      Initializing 1 input bytes: [0]=29
10:   SetGuestEffectiveAddress returned 0020C490

11: Write Mem Access at address 001EF987 of 1 bytes
12:   SetGuestEffectiveAddress returned 001EF987

13: END: ExitProcess is called

14: ***** External Memory Stats: *****
15: Number of Mem Accesses: 2 (2 Reads, 0 Writes)
16: Number of Addresses: 2 (total 5 bytes)
17: Number of Inputs: 2 (total 5 bytes)

18: ***** Native Memory Stats: *****
19: Number of Module Accesses: 0 (0 Reads, 0 Writes)
20: Number of Other Accesses: 1 (0 Reads, 1 Writes)

21: ***** General Stats: *****
22: Number of Unique Instructions After Start: 9
23: Number of Warnings: 0
24: Number of Errors: 0
```

# Overview

- Report of Micro Execution

```
mov    ebp, esp
push   ecx
mov    eax, DWORD PTR [ebp+8]  ; p
mov    cl, BYTE PTR [eax]      ; *p
mov    BYTE PTR [ebp-1], cl    ; v
```

  - line 11, memory access [ebp-1]
    - ebp-1 = 0x001EF987 is not input
    - is local variable
    - is mapped to itself
    - [0x001EF987] = 0x29

```
1:   initEIP is 72B51005
2:   initEBP is 001EF988

3:   Read Mem Access at address 001EF990 of 4 bytes
4:       Initializing 4 input bytes:
5:         [0]=78 [1]=14 [2]=20 [3]=00
6:       Adding 00201478 to list of known addresses
7:     SetGuestEffectiveAddress returned 00201440

8:   Read Mem Access at address 00201478 of 1 bytes
9:       Initializing 1 input bytes: [0]=29
10:    SetGuestEffectiveAddress returned 0020C490

11:  Write Mem Access at address 001EF987 of 1 bytes
12:    SetGuestEffectiveAddress returned 001EF987

13:  END: ExitProcess is called

14:  ***** External Memory Stats: *****
15:  Number of Mem Accesses: 2 (2 Reads, 0 Writes)
16:  Number of Addresses: 2 (total 5 bytes)
17:  Number of Inputs: 2 (total 5 bytes)

18:  ***** Native Memory Stats: *****
19:  Number of Module Accesses: 0 (0 Reads, 0 Writes)
20:  Number of Other Accesses: 1 (0 Reads, 1 Writes)

21:  ***** General Stats: *****
22:  Number of Unique Instructions After Start: 9
23:  Number of Warnings: 0
24:  Number of Errors: 0
```

# How is MicroX implemented?

- Program Instrumentation
  - 임의의 코드를 수행하면 할당되지 않은 메모리에 접근한 후 얼마 지나지 않아 충돌이 발생
  - VM이 메모리를 접근하는 명령어를 실행하기 전에 무작위로 할당해주기 때문에 충돌이 발생하지 않게됨.
  - MicroX VM은 x86 instruction을 micro-operation 시퀀스로 분해함.

```
ex) mov eax, [ecx]
    (1) use the 32-bit value of ecx as an address
    (2) get the 32-bit value located at that address (denoted [ecx])
    (3) copy that value in eax

   => this instruction is rewritten as a sequence of micro-operations
      (GenerateEffectiveAddress, PREMemoryAccessCallBack etc.)
```

# How is MicroX implemented?

- Program Instrumentation
  - GenerateEffectiveAddress is an x86 macro
    - 현재 명령어에 엑세스 할 주소를 계산
  - PREMemoryAccessCallBack is a new MicroX callback
    - 메모리 접근 명령어를 수행하기 전에 접근할 메모리를 할당/계산
    - 접근할 메모리를 그대로 놔둬야할지 (A), 새로운 것으로 대체해야 할지 결정 (B)
      - external memory와 local variable의 차이

```
ex) mov eax, [ecx] ➔ mov eax, [EffectiveAddress]
    the code under test is never aware of EffectiveAddress
```

# How is MicroX implemented?

- Program Instrumentation
  - decomposing x86 instructions into sequences of micro-operations is a *standard technique for dynamic binary program instrumentation* [3]

  - PREMemoryAccessCallBack is new, to the best of our knowledge (e.g., compared to Nirvana [3] or PIN)

# How is MicroX implemented?

- External Memory Management
  - maintains a mapping from program-visible addresses to External Memory addresses.
    - this mapping is used to ensure read/write memory consistency (ex. address a에 있는 값이 v라면, 다음에 다시 읽을 때도 v이어야 함.)

# How is MicroX implemented?

- External Memory Management

```
int r; // Heap_Address_Range; default 250
int r_EBP;  // EBP_Address_Range; default 100
int InitEBP; // initial value of EBP
list_of_ADDR KnownInputAddresses;

bool IsInputAddress(ADDR a) {
  if ((InitEBP <= a) && (a < (InitEBP + r_EBP)))
     return true;
  for any x in KnownInputAddresses {
    if (((x >= a) && ((x - a) < r))
       || ((x < a) && ((a - x) < r)))
       return true;
  }
  return false;
}
```

```
ADDR PREMemoryAccessCallback
(ADDR a, int size, bool isRead)
{
  ADDR a' = ExternalMemory(a);
  if (a' is defined) return a';
  if (!IsInputAddress(a)) return a;
  Add ADDR [a,a+size-1] to ExternalMemory;
  a' = ExternalMemory(a);
  if (isRead) {
    initialize the values at ADDR [a,a+size-1]
      in ExternalMemory;
    if (size == 4)
      { add the 4-byte value at [a,a+3]
         to KnownInputAddresses; }
  }
  return a';
}
```

# How is MicroX implemented?

- Input Value Generation
  - Input values can be generated using many different way
    - Zero mode : all input values are zero (useful for debugging)
    - Random mode : all input values are randomly generated
      - 충돌하지 않도록 설계
    - File mode : input values are read from a file (reusable)
    - Process-dump mode : initial input values are read from a process dump (using windbg debugger)
    - SAGE mode : it is good for case of the presence of pointer arithmetic in the code.
      - It implements the precise pointer-reasoning algorithms

# How is MicroX implemented?

- Other Implementation Details
  - Nirvana
    - a dynamic binary rewriting tool
    - translate into a sequence of micro-operations
  - iDNA
    - is a Nirvana application records binary execution traces
    - can be replayed in a debugger (like windbg)

# How is MicroX implemented?

- Other Implementation Details
  - TruScan
    - takes as input an iDNA trace and analyzes it for various properties (BOFs, memory leaks, uninitialized variables, etc.)
    - 그대로 사용함.
  - SAGE
    - is an automatic test generation tool using dynamic symbolic execution

# Limitations of Micro Execution

- False positives (too many behaviors, unrealistic bugs)
  1. the input set is too large (too many inputs)
  2. the set of possible input values is too large (too many input values)
  - 적절하게 input 의 범위를 줄일 수 있어야 함.

- False negatives (too few behaviors, missed bugs)
  - It means that test coverage is low

# Experimental Results

| Function Name | Unique Instructions (avg [min-max]) | Inputs (avg [min-max]) | Memory Accesses (avg [min-max]) | Tests | Crashes |
|---|---|---|---|---|---|
| _i64toa_s | 179 [124-211] | 5 [5-5] | 202 [69-323] | 23 | 0 |
| _snwscanf_s | 164 [76-388] | 5 [1-7] | 60 [23-155] | 18 | 0 |
| _splitpath_s | 142 [142-142] | 89 [37-221] | 431 [170-1090] | 4 | 0 |
| _strnset_s | 82 [48-139] | 74 [3-215] | 201 [8-636] | 10 | 0 |
| _strset_s | 81 [30-128] | 27 [1-253] | 105 [4-754] | 56 | 0 |
| _ui64toa_s | 165 [121-208] | 5 [5-5] | 242 [68-753] | 19 | 0 |
| _ui64tow_s | 169 [121-209] | 5 [5-5] | 258 [68-1105] | 18 | 0 |
| _ultoa_s | 107 [67-164] | 36 [4-502] | 121 [20-1026] | 31 | 2 |
| _ultow_s | 119 [74-167] | 25 [4-252] | 107 [22-529] | 23 | 2 |
| _vsnprintf_s | 222 [116-275] | 34 [3-101] | 660 [66-2030] | 24 | 0 |
| _i64tow_s | 181 [124-212] | 5 [5-5] | 199 [69-319] | 21 | 0 |
| _vsnwprintf_s | 144 [139-153] | 90 [7-130] | 2172 [59-3189] | 6 | 6 |
| _wcsnset_s | 79 [36-141] | 57 [2-378] | 1691 [5-100000] | 66 | 4 |

# Experimental Results

- Unique Instructions
  - Micro Execution을 수행할 때 다양한 경로를 실행한 것을 확인
- Inputs
  - 일정한 수치로 나타나는 것들은 숫자를 입력 받음.
  - 나머지는 String 처리 함수 (다양한 길이의 input 발생)
- Memory Accesses
  - 모든 input들이 memory access read를 포함하기 때문에 당연한 결과
  - _wcsnset_s 함수는 메모리 접근 제한 횟수까지 도달함.

# Experimental Results

- Tests
  - 1분 동안 Micro Execution 한 횟수
  - _splitpath_s 함수는 142 unique instructions에도 불구하고 오래걸림
    - SAGE 에서 symbolic execution 을 수행하는데 오래걸림.

- Crashes
  - 실험 대상 함수 이름에 _s 접미사가 있음.
    - 입력 파라미터를 생성할 때, 유효성 검사나 보안을 체크해야함
  - 대부분 0으로 나타난 것으로 보아 MicroX와 SAGE를 통해 쉽게 처리할 수 있는 정도임.

# E N D